DOCUMENT RESUME

ED 321 736 IR 014 498

AUTHOR Pieters, Jules M., Ed.
TITLE Instructional Aspects of Intelligent Tutoring Systems.
INSTITUTION Twente Univ. of Technology, Enschede (Netherlands).
REPORT NO ISBN-90-365-0298-5
PUB DATE 89
NOTE 82p.
PUB TYPE Collected Works - General (020) -- Reports - Research/Technical (143)

EDRS PRICE MF01/PC04 Plus Postage.
DESCRIPTORS Cognitive Processes; *Computer Assisted Instruction; *Expert Systems; Foreign Countries; *Grammar; Interaction; *Problem Solving; *Programing; *Spelling Instruction
IDENTIFIERS *Intelligent Tutoring Systems

ABSTRACT

This collection contains three papers addressing the instructional aspects of intelligent tutoring systems (ITS): (1) "Some Experiences with Two Intelligent Tutoring Systems for Teaching Computer Programming: Proust and the LISP-Tutor" (van den Berg, Merrienboer, and Maaswinkel); (2) "Some Issues on the Construction of Cooperative ITS" (Kanselaar, Andriessen, Barnard, and Erkens); and (3) "Artificial Intelligent Tools for Grammar and Spelling Instruction" (Pijls, Daelemans, and Kempen). Each of the papers includes refrences. (GL)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

ED321736

Instructional Aspects of Intelligent Tutoring Systems

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

☒ This document has been reproduced as received from the person or organization originating it.

☐ Minor changes have been made to improve reproduction quality.

• Points of view or opinions stated in this document do not necessarily represent official OERI position or policy.

Jules M. Pieters (Ed.)

Onderzoeksthemagroep
Onderwijsleerprocessen

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY

J. Nelissen

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

IR014492

toegepaste
ONDERWIJSKUNDE

Universiteit Twente

BEST COPY AVAILABLE

INSTRUCTIONAL ASPECTS OF INTELLIGENT TUTORING SYSTEMS

Jules M. Pieters (Ed.)

ONDERZOEKSTHEMAGROEP
ONDERWIJSLEERPROCESSEN

Instructional aspects of intelligent tutoring systems/Jules M. Pieters (Ed.)-Enschede: Universiteit Twente, Onderzoeksthemagroep Onderwijsleerprocessen, 1989.
ISBN 90-365-0298-5

Trefwoorden: instructietechnologie
computerondersteunde instructie
intelligente tutoriële systemen

Colofon:
Vormgeving: AV-sectie Toegepaste Onderwijskunde
Reproductie: Centrale Reproductie-afdeling Universiteit Twente
Oplage: 75
Besteladres: Universiteit Twente
Mediatheek TO
Postbus 217
7500 AE Enschede
Tel.: 053-893588

(c) 1989 Alle rechten voorbehouden.

Niets uit deze uitgave mag worden verveelvuldigd en/of openbaar gemaakt worden door middel van druk, fotocopie, microfilm of op welke wijze dan ook, zonder voorafgaande schriftelijke toestemming van de auteurs.

No part of the material protected by this copy right notice may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without written permission from the copyright owner.

CONTENTS

page

1. INTRODUCTION 5
J.M. Pieters
University of Twente

2. SOME EXPERIENCES WITH TWO INTELLIGENT TUTORING SYSTEMS FOR TEACHING COMPUTER PROGRAMMING: PROUST AND THE LIPS-TUTOR 7
J. van Merriënboer
K.G. van den Berg
D.M. Maaswinkel
University of Twente

3. SOME ISSUES ON THE CONSTRUCTION OF COOPERATIVE ITS 45
G. Kanselaar
J.E.B. Andriessen
Y.F. Barnard
G. Erkens
University of Utrecht

4. ARTIFICIAL INTELLIGENT TOOLS FOR GRAMMAR AND SPELLING INSTRUCTION 67
F. Pijls
W. Daelemans
G. Kempen
University of Nijmegen
University of Brussels

PREFACE

How intelligent are intelligent non-human systems is a question that is often raised since endeavours have been undertaken to simulate human intelligence. In early days artificial intelligent behavior was expressed by LISA who could operate as a psychotherapist and in the seventies Newell & Simon tried to capture intelligence within the framework of human problem solving. In recent years, due to great progress that has been made in creating computerized systems that exceeds human notion of high speed computation and large storage capacities, sophisticated models of human intelligent behavior have been designed. Of course, developments in cognitive psychology facilitates thinking about intelligence and likewise creating these models. Artificial intelligence grew older and became a child that could talk and discuss things with her creators instead of reacting passively to stimuli presented by him. It became technically possible to set up a dialogue that resembles a discussion between two intelligent beings. So it even became possible to let and play one of them the role of a teacher, artificially however. Learning programs evolved from inflexible presentation and very limited diagnostic and feedback facilities to programs that understood reactions from the student and even could answer appropriately. Cognitive psychology put forward models that simulate knowledge acquisition and knowledge representation. And this development was very important for designing intelligent systems. Now, artificial systems could rely on an internal diagnostic expert and a subject matter expert.

In the aforementioned models these components can be discerned. Each component serves a part of the intelligent computations made by an artificial intelligent system. In their contribution to this volume, Jeroen van Merriënboer, Klaas van den Berg and Dolf Maaswinkel picture the development of these components in a number of famous intelligent systems.

However, the perhaps most important component of all, the tutoring component, being the intermediate mechanism that presents subject matter in a, instructional technologically

speaking, proper way to the learner, was (and probably is) hard to design. Recent research efforts have been directed to this Achilles heel of intelligent tutoring systems.

In June 1988 the OTG-Onderwijsleerprocessen organized a meeting in which instructional aspects of intelligent tutoring systems were discussed. In this volume three contributions to that meeting are presented.

Jeroen van Merriënboer, Klaas van den Berg and Dolf Maaswinkel discuss two intelligent tutoring systems for teaching introductory programming.

Gellof Kanselaar, Jerry Andriessen, Yvonne Barnard and Gijsbert Erkens discuss a cooperative paradigm for intelligent tutoring systems.

Fieny Pijls, Walter Daelemans and Gerard Kempen describe the development of an intelligent computational instruction environment for grammar teaching. This contribution was also released as an article in Instructional Science.

J.M. Pieters

SOME EXPERIENCES WITH TWO INTELLIGENT TUTORING SYSTEMS FOR TEACHING COMPUTER PROGRAMMING: PROUST AND THE LISP-TUTOR

Jeroen J.G. van Merriënboer
Klaas G. van den Berg
Dolf M. Maaswinkel

University of Twente

Abstract
In this article, we report on our experiences with PROUST and the LISP-Tutor, which are intelligent tutoring systems (ITS) for teaching introductory programming in, respectively, Pascal and LISP. After a brief discussion of the history of intelligent programming tutors, and the presentation of a general framework for discussing the architecture of such systems, the functional architecture and the performance of PROUST and the LISP-Tutor are described. Then, we are in a position for a critical evaluation and comparison of the two systems. Finally, it is discussed what the strengths and weaknesses of PROUST, the LISP-Tutor, and more recent systems inspired by them may learn us for the development of new programming tutors.

1. INTRODUCTION

In the summer of 1984, the Interfacultary Working Group Teacher Education at the University of Twente founded the DISCS research group to study the design of instruction in the field of computer science (Designing Instructional Systems for Computer Science; Dijkstra, Krammer, Maaswinkel, Van Merriënboer, Van Den Berg, Van Dijk, & Duijvestijn, 1988). Several researchers in this group are interested in programming instruction for high- level languages, and more in particular, in the design of introductory programming courses at high school and university levels.

As Intelligent Tutoring Systems (ITS) for teaching programming may be especially helpful in introductory courses, in 1987 the Study Group Intelligent Programming Tutors (StIPT) was founded. This group has been studying and evaluating two well-known programming tutors: The LISP-Tutor, for teaching elementary LISP, designed by the group of J.R. Anderson at Carnegy Mellon University (CMU) in Pittsburgh, and PROUST (PROgram Understander for STudents), for debugging beginner's Pascal programs, designed by the group of E. Soloway at Yale University in New Haven. The goal of this article is to discuss their strengths and weaknesses, to compare their architectures, to communicate our experiences with them, to report some recent developments concerning the systems, and, finally, to present some options to design a programming tutor that capitalizes on the strengths of both systems.

The approach of StIPT to the study of programming tutors is inspired by our philosophy about teaching and learning program design. In fact, we have been promoting an instructional strategy for beginning programmers that is known as the 'reading approach' or the 'completion strategy' (Van Merriënboer, 1988, in press; Van Merriënboer & Krammer, 1987). According to this strategy, there must be four phases in learning program design. In the first phase, students run working programs, observe their behavior, and evaluate their strengths and weaknesses. In phase two, students are actually introduced to the programs; their primary activities in this phase are reading and hand tracing the programs to gain insight in their architecture. During the third phase, students modify or amplify the programs; and finally, they must be able to construct programs on their own. As our studygroup mainly consisted of 'beginners' in the field of programming tutors, we thought it would be nice to adhere to this approach.

Consequently, we laid hands on two programming tutors (PROUST and the LISP-Tutor), and we tried to run them to observe their behavior and study their strengths and weaknesses. This was not an easy task: The programs were hardly portable from one computer system to another because they are not very well documented and heavily rely on specific systems software, so it took us considerable time to get them running. Then we were able to start reading and tracing the systems to get a clear view of their architectures. This was

also a time-consuming task because both systems include approximately 20-thousand lines of source code. Nevertheless, we recently completed this task and we are now planning modifications and extensions to the systems (Stijnen & Offereins, 1988). In the final phase, we will use the acquired knowledge to construct our own programming tutor.

In this article, we report on our experiences during the first three phases. The structure of the paper is as follows. To provide the reader with some background, Chapter 2 offers a brief history of programming tutors. In Chapter 3, a generalized structure for ITS's is presented that may subsequently be used as a framework to discuss the architecture of PROUST (Chapter 4) and the LISP-Tutor (Chapter 5). In Chapter 6, we are in a position to thoroughly compare the two systems. Chapter 7 discusses some recent developments in programming tutors that are inspired by either PROUST or the LISP-Tutor. Finally, Chapter 8 provides the reader with a summary and discusses some options to design a system that capitalizes on the strengths of both PROUST and the LISP-Tutor.

2. A BRIEF HISTORY OF PROGRAMMING TUTORS

A brief, far-from-complete history of programming tutors, or, more generally, systems for assisting in program design and analysis, is displayed in Table 1. The table is restricted to well-known, representative systems. In the columns appear, in order, the year of appearance, the name of the system, the name of the first author, and the language that it is tutoring. As may be seen from the table, the history of programming tutors can be traced back to 1974, when the Stanford Basic Instructional Program appeared (BIP; Barr, Beard, & Atkinson, 1975, 1976). As early as in 1976, the term 'intelligent' was used in relation to a system for analyzing PL/1 programs (IPA; Ruth, 1976).

Most systems that were developed in the 1980ties are meant for tutoring either Pascal or LISP. Early systems for tutoring Pascal are PUDSY (Lukey, 1980) and the Pascal Assistant (PA; Singer, Ledgard, & Hueras, 1981). In 1982, a forerunner of PROUST appeared under the name MENO-II (Soloway, Woolf, Barth,

Table 1. A Brief History of Programming Tutors

Year	Name	Author	Language
1974	BIP BASIC Instructional Program	Barr	Basic
1976	IPA Intelligent Program Analysis	Ruth	PL/1
1980	LAURA	Adam	Fortran
1980	PUDSY Program Understanding and Debugging System	Lukey	Pascal
1981	PA Pascal Assistant	Singer	Pascal
1982	MENO-II	Soloway	Pascal
1982	SPADE-0	Miller	Logo
1982	PHENARETE	Wertz	LISP
1983	PROUST Program Understander for Students	Johnson	Pascal
1984	GREATERP Goal-restricted environment for tutoring and educational research on programming	Anderson	LISP
1985	TALUS	Murray	LISP
1986	RADAR Reasoning on ADA Rubbish	Cerri	ADA
1986	AURAC	Hasemer	Solo
1987	BRIDGE	Bonar	Pascal
1988	CHIRON	Sack	Pascal
1988	GIL Graphical Instruction in LISP	Reiser	LISP

& Rubin, 1981; Soloway, Rubin, Woolf, & Bonar, 1984). The first version of PROUST appeared in 1983 (Johnson, 1985, 1986; Johnson & Soloway, 1985a, 1985b; Sack, Littman, Spohrer, Lilies, Fertig, Hughes, Johnson, & Soloway, 1986); more recent Pascal programming tutors that stem from the Yale-school and that are still heavily inspired by PROUST are, for instance, BRIDGE (Bonar, Cunningham, Beatty, & Riggs, 1987) and CHIRON (Sack, 1988; Sack, Littman, Spohrer, Fertig, Hughes, & Soloway, 1988).

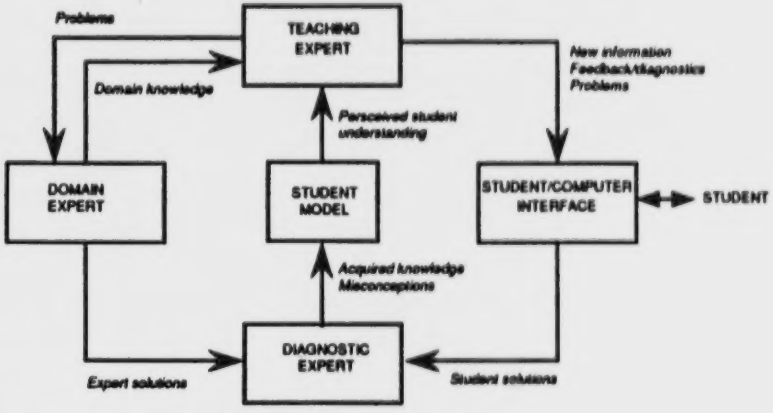
The first version of the LISP-Tutor appeared in 1984 under the name GREATERP, which stands for Goal Restricted Environment for Tutoring and Educational Research on Programming (Reiser, Anderson, & Farrell, 1985). Probably for sake of simplicity, later versions of the system became known as the LISP-Tutor (Anderson & Reiser, 1985; Anderson & Skwarecki, 1986); a more recently developed system for tutoring LISP is GIL (Reiser, Friedmann, Kimberg, & Ranney, 1988; Reiser, Friedmann, Gevins, Kimberg, Ranney, & Romero, 1988). GIL stems from the CMU-school; whereas it is graphically oriented, like BRIDGE, it is still using the main ideas from the LISP-Tutor.

What can we say about the development of programming tutors on the basis of this brief history? First, that programming tutors using AI techniques have been existing for over 15 years, although the term ITS, or the term intelligent programming tutors, is quite new. Second, that most programming tutors of the last decade are developed for tutoring either Pascal or LISP. With regard to Pascal, the Yale-school seems most influential; with regard to LISP, the CMU-school seems most influential: Recent systems for tutoring Pascal or LISP are often heavily inspired by the approach of either PROUST or the LISP-Tutor. PROUST and the LISP-Tutor themselves are no longer under substantial development; in fact, the authors feel that those systems may be conceived of as the outturn of the first decade of research on intelligent programming tutors. For this reason, it now seems the right time for a critical evaluation of the systems. In the following chapter, a generalized structure for ITS is presented that subsequently will be used as a framework for such an evaluation.

3. A GENERALIZED INTELLIGENT TUTORING SYSTEM

Wallach (1987) presented a generalized structure for intelligent tutoring systems containing the major components of typical ITS's. Figure 1 provides a simplified version of his generalized program, containing only the components that are necessary for our following discussion. As may be seen in the figure, a typical ITS contains most or all of the following five functional components: A domain expert, a diagnostic expert, a student model, a teaching expert, and, finally, a student-computer interface.

Figure 1. A Generalized Structure for Intelligent Tutoring Systems (Adapted from Wallach, 1987).



The domain expert contains the knowledge about the field that the students need to learn. This usually is both procedural and factual knowledge so that the domain expert is able to solve the same problems as the students have to solve. In the ideal case, those problems are presented to both the domain expert and the students by the teaching expert. The domain expert generates solutions that provide a basis for comparison to the students' solutions by the diagnostic expert. The solutions of the domain expert are often called 'expert solutions'; however, in many cases it seems more appropriate

to talk about 'ideal student solutions'. In addition to solving problems that are generated by the teaching expert, the knowledge of the domain expert may also be directly used by this teaching expert. In case that the teaching expert is presenting new information to the students or giving feedback on their incorrect solutions, it may take this knowledge from the domain expert.

The diagnostic expert mainly compares expert solutions, or ideal student solutions, to actual student solutions. Thus, it uses rules and 'ideal' solutions to analyze student responses; based on those responses, the system acquires a certain understanding of what the student does and does not know. This model of the student's understanding is usually comprised in a student model. Generally, two aspects can be distinguished with regard to this model. First, it represents what the student already knows; this may be called the 'acquired knowledge'. Second, the model represents what the student does not understand; this misunderstandings may be called 'misconceptions'. Obviously, the student model is a central part of the complete system as most actual tutoring will be based on this model of the student.

The student model forms the input to the teaching expert, which possesses knowledge about teaching and instruction. Based on the model of the student, its job is to choose the next strategy for teaching the student. Usually, several options are available. If the student solution is containing errors, that is, if this solution is not identical to the expert solution because of misconceptions, it can provide feedback or diagnostics on those errors. As a second example, in case of incomplete knowledge or a lack of acquired knowledge, it can present new information to the student. Finally, the teaching expert may select new problems to present to the student and, sometimes, to the domain expert.

Each of the experts consists of knowledge bases and an executable program ('inference engine'), which uses the information in the knowledge bases to arrive at conclusions. The knowledge may be represented in many ways; representations that are tailored for a specific domain are often inadequate for other domains so that, usually, different knowledge representations are required for each expert. In addition, there are a variety of methods for searching through the knowledge bases, depending upon the knowledge representation

and the purpose of the expert. For example, the backward chaining technique begins with a hypothesis that is to be proven or unproven; it works backwards through the knowledge base looking for rules that can prove the hypothesis. It often matches the requirements of the teaching expert because this has the specific goal of deciding what to do next; in contrast, it seldom matches the requirements of the domain expert because this has the more general goal to derive all possible 'ideal' solutions.

As a final module, there is an interface between the computer and the student. Often, a high-quality student interface is needed containing graphics or restricted natural language, because it is usually necessary to gather precise information about the student's problem solving processes. So far our sketch of a generalized ITS. In the following sections, we will use this generalized structure as a framework for discussing PROUST and the LISP-Tutor.

4. PROUST'S ARCHITECTURE

In one of the early publications on PROUST (Johnson & Soloway, 1985b) the system was announced as "An automatic debugger for Pascal programs". This is exactly what PROUST is: It is a system that is able to identify multiple bugs in Pascal programs; it provides tutorial feedback on programs with--non-syntactical--bugs, trying to catch as many bugs as possible in one analysis. We will describe the several experts present in PROUST in a step- by-step way, referring to the generalized ITS-structure that was introduced in the previous chapter.

PROUST's Domain Expert

PROUST's domain expert is actually quite simple with regard to its relations with other experts. There is no communication with the teaching expert: Its domain knowledge is not used by the teaching expert, and the problems it has to solve are not generated by the teaching expert. Instead, the domain expert has a database with descriptions of problems it can solve. These problem descriptions are not stated in something like natural language; in fact it takes a programmer with thorough

knowledge of the internal structure of PROUST to extend the system for handling new problems.

The type of analysis that is performed is goal-plan analysis. Each problem in the database is stated in terms of goals that have to be reached in order to solve the problem. The domain expert has a knowledge-base that describes each goal in terms of plans. A plan is a template of language code or a pattern of stereotypic sequences of computer instructions to reach a goal; furthermore a plan may contain other goals, or subgoals, that have to be reached in order to make the plan satisfying the original goal. Plans are (partial) solutions on a higher level of abstraction than the actual program code. This facilitates reasoning on programs and the constituting parts of it. The expert solutions generated by the domain expert consist of compositions of such plans. To elaborate a little further on the goal-plan analysis, consider the following example. The problem to be solved is stated as:

Write a program which reads in a sequence of numbers, stopping when 99999 is read. Compute the average of these numbers. Do not include 99999 in the average.

This problem is known as the average problem; the stopping value 99999 is called the sentinel. Although it is a very simple problem, it is complicated enough to illustrate the goal-plan analysis. The problem description as it appears in the database of the domain expert is written as:

```
DefProgram  Average;
DefObject   ?Avg:Input;
DefGoal     Sentinel-Controlled-Input-Sequence( ?Avg:Input,
              99999);
DefGoal     Output(Average(?Avg:Input));
```

This cryptic description states that the program is to handle a list of data: ?Avg:Input. Two goals must be reached. The first goal is a sentinel- controlled-input-sequence, reading the list ?Avg:Input until the stopping value 99999 is reached. The second goal is a compound goal; the goal is to output the result of the goal Average(?Avg:Input), that is, the average of the list of numbers read.

Figure 2. A Simplified Goal-Plan Analysis for the Average Problem.

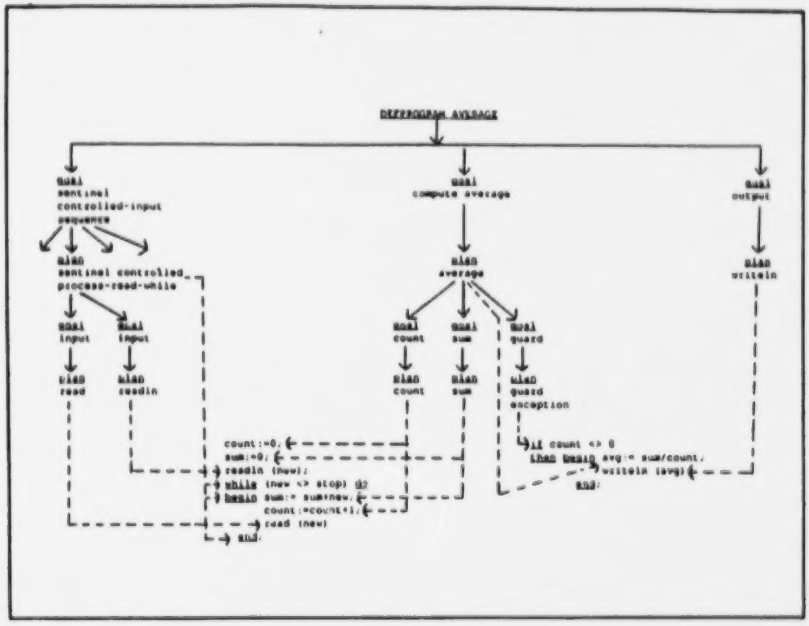


Figure 2 shows an example of a simplified goal-plan analysis for this problem, together with a sample program. The main goal is to write an average program. Three subgoals can immediately be determined from the problem description:
- read numbers until the sentinel is encountered (goal sentinel-controlled- input-sequence);
- compute the average;
- output the result.

There are many ways to reach the goal sentinel-controlled-input-sequence. In Figure 2 only the sentinel-controlled-process-read-while-plan is shown. As the other outgoing branches suggest, there are many more plans to implement this goal. This applies to all goals, but alternative plans are left out of the example for reasons of

clarity. Two subgoals must be reached to implement a sentinel-controlled-process-read-while-plan:
- input before the while-loop;
- input inside the while-loop.

Here it can be seen, that a goal can be reached by a plan, that requires that other goals, or subgoals, are satisfied: This is the general idea in goal-plan analysis. More complicated possibilities of plan descriptions are visible in the following average plan:

```
VARIABLES:
          ?Avg, ?Sum, ?Count, ?New
PRIOR GOALS:
          Count(?New, ?Count)
          Sum(?New, ?Sum)
POSTERIOR GOALS:
          Guard Exception(component Update: of Goal
                Average,
          ((?Count from goal Count) = 0))
EXCEPTION CONDITION:
          ((?Count from goal Count) = 0)
TEMPLATE:
          (component Mainloop: of goal Sentinel-Controlled-
        Input-Sequence)
          followed by:
          Update: ?Avg := (?Sum / ?Count)
```

We will not fully explain this plan description, but some details of it can be seen in the goal-plan analysis of Figure 2. This plan has subgoals and a code-template for matching the program code for the computation of the average; furthermore it has references to the plan that reaches the sentinel-controlled-input-sequence-goal, as indicated by the dotted lines in Figure 2. This kind of context information is necessary to determine which variable is used for the accumulative sum, and which variable counts the numbers that are read in. Here we can see why plans also contain information about goals that have to be priory met, namely, in order to be able to refer to plan components. Likewise there are posterior goals. Prior and posterior goals provide a means of cross-referencing between plans.

In fact, the actual analysis is much more complicated. There is one other important feature to be mentioned here. PROUST has so-called analysis coordination rules. These enable the domain expert to reformulate and transform goals and plans to semantically equivalent ones. Such rules provide a more dynamic goal-plan analysis, and make it possible to handle a greater variability in programs.

PROUST's Diagnostic Expert

The basic function of the diagnostic expert is plan-matching. In plan-matching, the student's program is compared to plan compositions of the domain expert. As we have seen, single (sub)goals can be reached by several plans; so many plan compositions form a correct solution to a problem, and many student solutions can be recognized as a correct program. In fact, the communication between the diagnostic expert and the domain expert is highly interactive. The domain expert generates a plan composition, which the diagnostic expert matches against the student's program. If mismatches are detected, the diagnostic expert asks the domain expert to try to make a different plan composition. This interaction finally ends with one of the two following results:
- for all goals to be reached, there are plan compositions that completely match the student's program; this is a perfect match.
- for some of the goals to be reached, no plan compositions could be made to match the student's program completely; in this case there are mismatches.

In case of a perfect match, nothing happens; the student program is recognized as correct. In case of mismatching, the mismatches are reported to the teaching expert. It is important to note here, that there is no updating of a student model in either case.

PROUST's Student Model

As we have seen in the description of the diagnostic expert, there is no updating of a student model. In fact, PROUST has no student model at all. However, it should be well possible to extend PROUST with facilities to record for each student which plans were used correctly, and which plans were in

error; in fact, this would result in a rather qualitative student model. Such an extension of the system could provide the teaching expert with valuable information for generating tutorial feedback.

PROUST's Teaching Expert

The teaching expert receives a report of mismatches, and uses rules to generate bug-reports. These rules are called bug rules, and they explain mismatches in terms of the goals that the student had, but that were not reached because of incorrect plans or the incorrect use of plans. The bug-reports that are generated by a bug rule explain the bugs that are detected; moreover, these bug-reports provide hints for debugging.

A central point in this type of analysis and bug reporting is that it is based on the intentions of the student. Goal-plan analyses expose how the students intend to solve the problem, and which plans were used; bug-reports use this information to explain where and why incorrect plans were used. For this reason PROUST's approach is called intention based diagnosis (Johnson, 1985, 1986). As an example, consider PROUST's output to an incorrect solution of the average problem in Figure 3.

For Pascal programmers, it is easy to see what goes wrong in this program. As the first bug-report explains there is a bug in line 11: It increments the variable NEW, instead of reading in a new value. The intention of the student is identified: "You were trying to use line 11 to read the next input value"; The bug is pointed out: "Incrementing NEW will not cause the next value to be read in"; and finally, a hint is given: "You need to use a READ statement here ...". Bug-reports 2 and 3 refer to the same bug; however, the plans they stem from are different. Report 2 is a result of the average plan: If no numbers are read, the program will try to perform a division by zero. Report 3 comes from the output plan: It makes no sense to output something that is undefined. It is particularly important to note here that PROUST is able to detect multiple bugs.

Figure 3. An Example of PROUST's output on an Incorrect Version of the Average Program.

```
   1   PROGRAM Average (input, output);
   2   VAR Sum, Count, New, Avg: REAL;
   3   BEGIN
   4      Sum:= 0;
   5      Count:= 0;
   6      Read (New);
   7      WHILE New <> 99999 DO
   8         BEGIN
   9            Sum:= Sum+New;
  10            Count:= Count+1;
  11            New:= New+1
  12         END;
  13      Avg:= Sum/Count;
  14      Writeln ('The average is ', Avg);
  15   END;


   PROUST output:


   1.  It appears that you were trying to use
       line 11 to read the next input value.
       Incrementing NEW will not case the
       next value to be read in. You need to
       use a READ statement here, such as you
       use in line 6.
   2.  You need a test to check that at least
       one valid data point has been input
       before line 13 is executed. The average
       will bomb when there is no input.
   3.  The average is undefined if there is no
       input but line 14 outputs it anyway.
       You should output the average only
       when there is something to compute
       the average of.
```

PROUST's Interface

In fact, there is very little to say about PROUST's interface. It is extremely simple. Students enter their--syntactical correct--programs in the system, and the system reports back on bugs that are found. However, there is a stripped-down version of PROUST with a better developed interface called Micro-PROUST (Johnson & Soloway, 1985c). This version is developed for the IBM PC, and does not have the full diagnostic capabilities of PROUST. This product is still under development, but it presents the feedback in colorful graphics and it provides the student with advanced editing facilities (Personal communication; see Van Merriënboer, 1987).

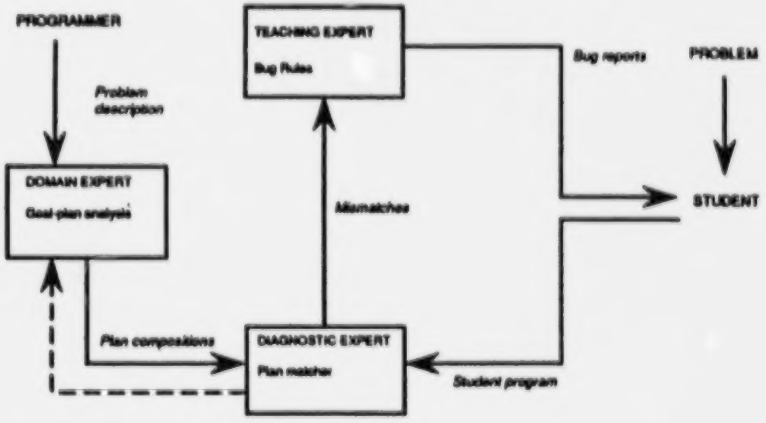
PROUST's Performance

More interesting is PROUST's performance. PROUST is set up to handle only a small number of programming problems, and as we have mentioned in the description of the domain expert, it is not easy to extend it with more problems. However, for the problems it is able to handle, it identifies a high percentage of 75% of the bugs correctly (Johnson, 1985, 1986). Furthermore, research on the educational effectiveness has shown, that students who had the benefit of PROUST's feedback performed significantly better in fixing bugs in their programming assignments than students who had no feedback from PROUST (Sack et al., 1986).

PROUST's Global Structure

At this point we are ready to compare PROUST with the generalized ITS- model as introduced in Chapter 3. Figure 4 presents PROUST's global structure; as becomes clear in this figure, PROUST does not implement all the functions.

Figure 4. PROUST's Global Structure.



A programmer has to enter the problem descriptions in the system. The domain expert uses goal-plan analysis to generate

possible solutions that have the form of plan compositions. The plan matcher is the heart of the system and is able to detect mismatches between expert solutions and the student's program. Mismatches are reported to the teaching expert, and this teaching expert has bug rules to explain the mismatches in terms of intentions that are not reached, because either incorrect plans were used or correct plans were used incorrectly.

PROUST has some strong and some weak points. The strongest feature is the diagnostic expert. The principle of intention based diagnosis offers students all freedom to compose their own solution. Even if they make a real mess of their programs, PROUST usually will still be able to generate feedback, because it can diagnose multiple bugs.

A weak point is the lack of communication between the domain expert and the teaching expert; no new problems are presented by the teaching expert, and the knowledge in the domain expert is not directly used in the tutoring. Furthermore, there is no student model, so that the system has no knowledge of what the student does or does not know. Finally, the interface between the student and the system is overly simple.

5. THE LISP-TUTOR'S ARCHITECTURE

The LISP-Tutor has been developed in the Advanced Computer Tutoring Project at the Carnegie-Mellon University. An article about this tutor appeared in the same april 1985 volume of Byte as about PROUST (Anderson & Reiser, 1985). Whereas the subtitle to the PROUST-article read 'An automatic debugger for Pascal programs', the subtitle to the LISP-Tutor stated 'It approaches the effectiveness of a human tutor'. The LISP-Tutor is much more interactive than PROUST. Other early publications on the LISP Tutor are given by Anderson, Farrell and Sauers (1984), Reiser, Anderson and Farrell (1985), and Anderson, Boyle and Reiser (1985). The design of the LISP-Tutor is based on the ACT* theory of cognition (Adaptive Control of Thought; Anderson, 1982, 1983, 1987a, 1987b). The assumptions of this theory are given by Corbett and Anderson (1988) and led to eight cognitive principles for the design of computer tutors (Anderson, Boyle, Farrell, & Reiser, 1987). We will describe

the LISP-Tutor and its five functional components, referring to the generalized ITS-structure.

The LISP-Tutor's Domain Expert

The LISP-Tutor's domain expert is more complicated than PROUST's domain expert because there is direct communication with the teaching expert. The problems the domain expert has to solve are received from the teaching expert and its domain knowledge is used by this teaching expert.

Instead of a system for goal-plan analysis, the domain expert consists of a production system modelling an ideal student. This system in the LISP- Tutor is called GRAPES (A loose acronym for 'Goal Restricted Production System'). The expert solutions generated by GRAPES can be seen as ideal solution paths, that are, the paths an ideal student should take to reach a correct solution to a problem. The basic concepts with regard to the domain expert of the LISP-Tutor are GRAPES, which is a set of productions or condition-action pairs to perform the programming task, and the solution path, which is a correct path in the goal tree that is generated by GRAPES. These concepts will be illustrated with an example.

Suppose that the following problem description is given to the domain expert (Anderson et al., 1984):

> Define a function called first-elem.
> Given any list, it returns the first element
> of that list. For example,
> (first-elem '(a b c)) returns a.

The correct solution to this problem in LISP code is:

```
(DEFUN first-elem (list1)
    (CAR list1))
```

DEFUN first-elem means that a function is defined with the name first- elem. The function has only one argument: A list with the name list1. On the second line the actual processing is given. CAR is a standard function that returns the head, that is the first element, of a list. The argument to this function is list1.

If GRAPES is solving a problem like this, or more complicated problems, it uses productions or condition-action pairs to generate an ideal solution path. Three simplified productions are given here:

```
P1    IF the goal is to write a function
   THEN write " ( DEFUN "
        and set subgoals to:
            1. Specify the name of the function;
            2. Specify the function arguments;
            3. Code the body of the function;
        and then write " ) ".

P2    IF the goal is to code the body of a function
        and the function recurses on a list
   THEN write " ( COND "
        and set subgoals to
            1. code the terminating cases;
            2. code the recursive cases;
        and then write " ) ".

P3    IF the goal is to add an element to a list
   THEN use CONS on the element and the list
        and set subgoals to
            1. Code the element;
            2. Code the list.
```

The first production P1 applies to the problem; the condition is satisfied to write a function, and the first action is to write " (DEFUN ", followed by the function name " first-elem " and the argument name " list1 ". The next subgoal is to code the process that is calculated by the function first-elem, and the final action is to write the closing parenthesis ") ".

In the LISP-Tutor there are 375 production rules in the ideal model. They are used to create ideal solution paths to given problems. Furthermore there are 475 buggy versions of these production rules. These will be discussed in the next section about the diagnostic expert. In the later versions of the LISP-Tutor a total of 1200 rules are used.

Recursion lessons in the LISP-Tutor are described by Pirolli (1985, 1986) and by Pirolli and Anderson (1985). A

more complicated problem with recursion is given by the following problem description:

> Define a function ADDTO, that given an element A, and a list with sublists Y, returns a list with lists with element A added to each of these sublists.
> For example:
> (ADDTO '3 '((5 8) (5) (8))) returns
> ((3 5 8) (3 5) (3 8))

A correct solution to this problem in LISP code is:

```
(DEFUN ADDTO (A Y)
   (COND ((NULL Y) NIL)
         (T (CONS (CONS A (CAR Y))
                  (ADDTO A (CDR Y))))))
```

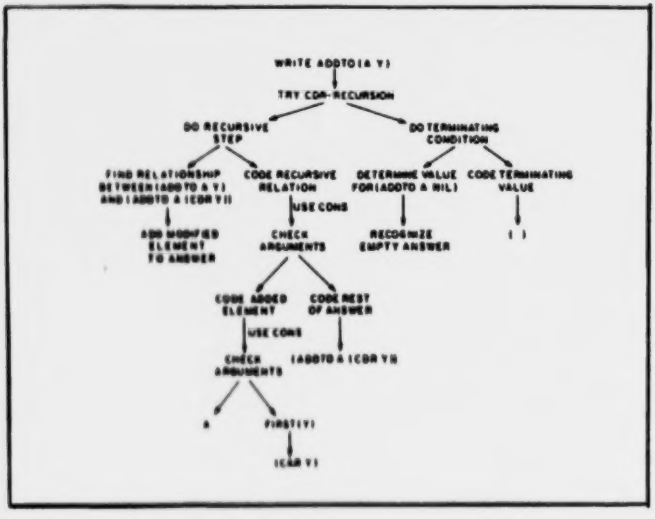
In this example, the second production P2--which involves recursion-- applies. Based on P2, P3 and other productions GRAPES will produce the hierarchical goal structure as given in Figure 5. There is certain resemblance between PROUST's goal-plan analysis and the LISP-Tutor's goal structure. The main difference lies in the fact that PROUST uses plans in its goal tree, which make it possible to reason on a level above the actual code of the program.

The LISP-Tutor's Diagnostic Expert

The diagnostic expert in the LISP-Tutor is based on the model tracing methodology, as derived from the ACT* theory (Anderson et al., 1988). In model tracing, the student's problem solving behavior is step-by-step compared to the model solution or the ideal solution path, as produced by the domain expert.

There are two possibilities. First, the problem solving behavior can be explained by ideal productions in GRAPES. These productions, that the student can use correctly, are added to the student model; the rules are seen as the knowledge that is already acquired by the student. In the second case, the student's problem solving behavior deviates from the ideal solution path. Usually, this deviation can be explained by one of the buggy production rules. If such buggy productions apply, they are also added to the student model.

Figure 5. The Goal Structure Controlling GRAPES' Solution to the Addto-problem (Adapted from Anderson, Farrell & Sauers, 1984).



The incorrect or buggy production rules that are used by the student are seen as student's misconceptions. Examples of buggy productions are:

Buggy rule P4 IF the goal is to specify the name of a function
THEN write " ' "
and write the function name.

The related text to this buggy production rule is:
"You do not need a quote for the name of the function you are defining. All you have to do is to write it down."

Buggy rule P5 IF the goal is to add an element to a list
THEN use LIST on the element and the list
and set subgoals to
1. Code the element
2. Code the list

The related text to this buggy production rule is: "You should add an element to a list, but LIST is not the right function. If you LIST together (a b c) and (x y z), for example, you will get ((a b c) (x y z)). LIST just wraps parentheses around its arguments."

There are approximately 475 buggy production rules in the LISP-Tutor. Summarizing, there are two basic concepts in the LISP-Tutor's diagnostic expert. Model tracing is used to follow the student's problem behavior, and buggy production rules are used to explain the deviations from the ideal solution path.

The LISP-Tutor's Student Model

The productions as discussed above are particularly important with respect to the student model. The student's model consists of: Acquired knowledge, in the form of a set of productions that the student already used correctly, and misconceptions, in the form of a set of buggy rules or incorrect productions that might explain a certain deviation from the ideal solution path.

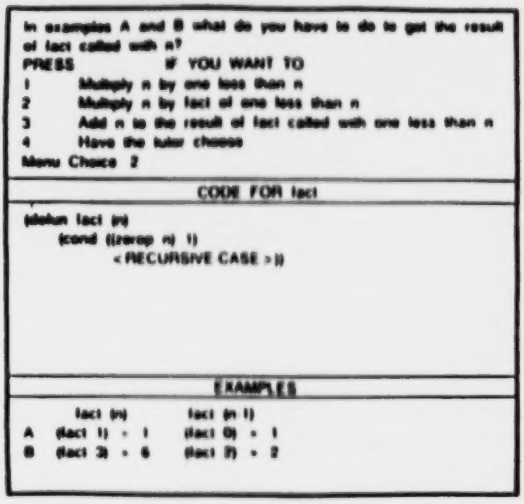
By frequently updating the student model during the use of the tutoring system by the student, it is possible to create a student history. The model of each student is essentially an "overlay" model that consists of a list of production rules in the ideal student model. For each rule the LISP-Tutor maintains an estimate of the probability that the student has learned the rule. The probability for a rule is updated when the student has the opportunity to employ the rule. The new estimate of whether the student has learned the rule depends on whether the student responds correctly or makes an error (Corbett & Anderson, 1988). Such a history will be used by the teaching expert to select its future actions, such as the sequence of remedial problems to accommodate the student's rate of progress.

The LISP-Tutor's Teaching Expert

The teaching expert receives its information from the student model and uses tutoring rules or t-rules to guide the tutoring process. The tutoring rules apply immediately when a student

makes an error. The t-rules can initiate several actions: They can present correct examples, if the student is thought to be able to map his or her solution to this correct example; they can present a diagnosis or present a hint, if the student is thought to be able to find the bug; or it can present the correct solution. In particular, the latter will be the case if no buggy production rule is found that might explain the deviation from the ideal solution path. As an example, Figure 6 presents a screen that is showing the actions of the tutor in guiding a student who is defining the factorial function.

Figure 6. An Example of the LISP-Tutor's Screen, Showing the Windows with Menu, Code and Examples.


In examples A and B what do you have to do to get the result of fact called with n?
PRESS IF YOU WANT TO
1 Multiply n by one less than n
2 Multiply n by fact of one less than n
3 Add n to the result of fact called with one less than n
4 Have the tutor choose
Menu Choice: 2
CODE FOR fact
(defun fact (n)
(cond ((zerop n) 1)
<RECURSIVE CASE>))
EXAMPLES
fact (n)
A (fact 1) = 1
B (fact 3) = 6
fact (n-1)
(fact 0) = 1
(fact 2) = 2

In contrast with PROUST, there is communication between the domain expert and the teaching expert in the LISP-Tutor. After solving a particular problem, new problems are selected on the basis of the student history. If productions have been correctly used for several times, the student will get a more difficult problem involving new productions. If only few productions are used correctly few times, the student will get another problem involving the same productions. The domain

expert will generate ideal solutions paths to these new problems.

The LISP-Tutor's Interface

The LISP-Tutor's interface is much more complicated than the interface of PROUST. Some of its most important principles will be discussed below.

Immediate feedback

Feedback is given immediately when a student makes an error and leaves the ideal solution path. This immediate feedback is one of the cognitive principles derived from the ACT* theory (Anderson et al., 1987) and implies that feedback should be presented right after an erroneous production is triggered. The actual symbol-by-symbol feedback in the present versions of the LISP-Tutor is tied to the grain-size of the productions in the student model (Corbett & Anderson, 1988).

Structured Editor

This editor provides the students with templates to enter their code. For example, if a student writes " (DEFUN ", the screen shows:

```
( DEFUN <NAME> <PARAMETERS>
        <PROCESS> )
```

The student can subsequently fill in the structure. See, for another example, also the code window in Figure 6.

Syntax checker

The goal of the syntax checker is to minimize students' processing load. Assistance in low-level tasks can help the students to devote more attention to higher level tasks, such as designing the solution. Moreover, it is known that syntax checkers do not lead to less knowledge of the syntax, but, on the contrary, to better knowledge of the syntax of a language.

Menu driven

Part of the interaction between the system and the student is menu driven in order to avoid complex natural language interfaces. These menu's are especially used to allow the student to communicate the steps of planning in the solution

of a problem. See, for an example, the upper window in Figure 6.

Highly reactive and interactive
The interface is highly reactive because of the (symbol-by-symbol) immediate feedback. There is intervention of the tutor as soon as the student makes an error. Furthermore, it is highly interactive because there is a lot of communication between the tutor and the student.

The LISP-Tutor's Performance

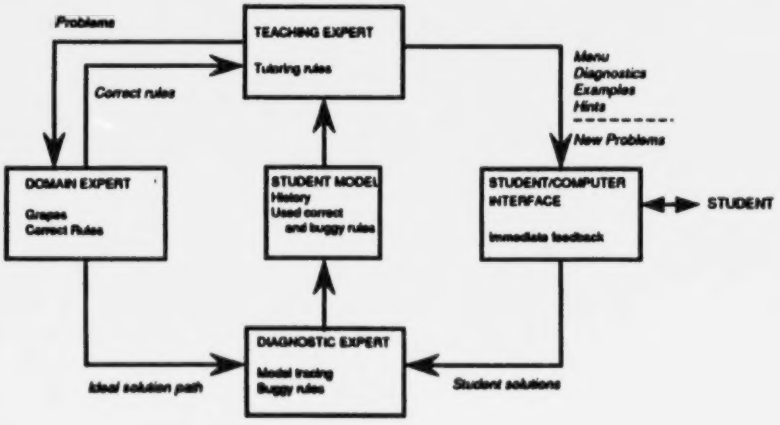
The LISP-Tutor is able to teach 12 lessons of approximately two hours each, covering all the basic concepts in LISP. The rules enable the tutor to diagnose and respond appropriately to between 45 and 80 percent of the students' errors, depending on the complexity of the lesson and the amount of testing of the lesson that has been conducted. In the well-worked lessons 80 percent of the student's errors are diagnosed correctly, while in the newest implemented lessons this is only about 45 percent. Students learned almost twice as quickly with the computer tutor as without a tutor, and nearly as quickly as those students with human tutors. Students working with the computer tutor performed better on the final examination than those working without the tutor (Reiser et al., 1985). At first sight, the rate of identified bugs mentioned above is about the same as in PROUST. But it is important to notice that in the LISP-Tutor only one bug can occur at a time, whereas PROUST is able to identify multiple bugs.

Another aspect of the LISP-Tutor's performance is in what degree it supports the underlying theory of cognition. This has been investigated in detail by Anderson (1986). Based on logfiles of the students' interaction with the tutor, he analyzed the time required the learn a production and the error rate in the production. The expected power function relating time or errors to amount of practice was confirmed in this study.

The LISP-Tutor's Global structure

The LISP-Tutor's global structure is presented in Figure 7. Problems enter the domain expert and are solved by GRAPES, which is a production system modelling an ideal student, that is able to generate ideal solution paths. In the model tracing methodology, the student's problem solving behavior is step-by-step compared to this solution path. In case of deviation from the path, buggy production rules can explain the deviation in most cases. The student model consists of those buggy production rules as well as the productions that the student used correctly. In case of the application of a buggy production rule, the tutoring rules present examples, diagnostics or hints. After solving a problem, new problems are selected on the basis of the student history. These problems are presented both to the student and to the domain expert, which provides new solution paths.

Figure 7. The LISP-Tutor's Global Structure.



One of the most elaborated features of the LISP-Tutor is its students model; the correctly used production rules and the used buggy production rules represent the student model and form the basis to the student history. This provides a sound basis for choosing tutorial strategies.

Heavily criticized is the principle of immediate feedback. The student has to follow the ideal solution path, and there is little room for the student to compose alternative solutions. The main reason for this is, that multiple bugs are not allowed. A second, related feature has to do with the overly interactive interface. The student is interrupted immediately after making an error, so that a lot of communication is needed between the tutor and the student. In fact, students do not only have little room to compose their own solutions, but they also have little freedom to correct their own errors. Especially more experienced students find this immediate feedback very annoying.

6. COMPARING PROUST AND THE LISP-TUTOR

In the previous chapters, the functional architectures of PROUST and the LISP-Tutor were described in terms of a generalized structure for ITS's. Now, we are in a position for a more direct comparison of the two systems. First, the systems will be compared with respect to their architectures; second, they will be briefly compared regarding their technical specifications.

Comparing the Architectures

In PROUST, the major task of the domain expert is performing goal-plan analyses to generate plan compositions. This type of analysis is quite powerful, because the use of plans makes it possible to reason about programs on a level that is above the actual code. Moreover, it offers the diagnostic expert the opportunity to recognize many student solutions as correct solutions. In the LISP-Tutor, the domain expert has the form a production system model of an ideal student that generates correct solution paths. This type of analysis is slightly less powerful, because it is more bound to the code level and, consequently, offers less opportunity to recognize different student solutions as correct programs.

With regard to the diagnostic expert, PROUST uses plan-matching to compare possible plan compositions with student solutions. The outcome of this process is a set of mismatches; therefore, it is especially suitable to diagnose multiple

bugs. In the LISP-Tutor, model-tracing is used to step-by-step compare actual student behavior with ideal student solutions. This process of model-tracing is especially suitable to cure one bug at a time by generating immediate instructional feedback; however, it is less suitable to generate feedback on multiple bugs.

With regard to the student model, large differences exist between PROUST and the LISP-Tutor. Whereas PROUST does not provide any kind of student model, in the LISP-Tutor it is quite sophisticated: It incorporates both the acquired knowledge, as a set of production rules that have been correctly applied, and the active misconceptions, as a set of buggy production rules that can explain deviations from the ideal solution path. Moreover, the student history is frequently updated so that the tutor is able to adapt itself to each individual student in next sessions.

In PROUST, the teaching expert contains bug rules that operate directly on the mismatches as reported by the diagnostic expert. These bug rules are able to explain the mismatches in terms of the missing or incorrect code that is responsible for not reaching the goals. No rules are available to choose instructional strategies, such as sequencing the instructional feedback or adapting the instruction to the individual students' needs. So, PROUST's general approach may be characterized as intention-based diagnosis of multiple bugs. In the LISP-Tutor, the teaching expert contains tutoring rules that apply if a buggy production is fired, that is, if the ideal path is left; the rules take the student model and student history into account. Sequencing the instructional feedback is unnecessary because only one bug at a time is cured, but instructional strategies are actually chosen on the basis of individual students' needs. Thus, the LISP-Tutors general approach may be characterized as individualized tutoring with immediate feedback on single bugs.

Finally, the system-student interfaces must be compared. PROUST's interface is overly simple: The student offers a to-be-analyzed program to the system and receives a bug-report in return. However, current work involves some interface-design on Micro-PROUST, which is a stripped version of PROUST for the PC. The LISP-Tutors interface is more complicated and contains an editing window with a structure-editor; however, it is quite inflexible because of its high overall interactivity as

well as reactivity in case of bugs. Here, it should be noted that the problems with immediate feedback have also been addressed in the Advanced Computer Tutoring Project (Anderson et al., 1988). In a recent version of the LISP-Tutor, there is a more flexible tutoring strategy, which includes the possibility of student controlled feedback (Corbett, Anderson, & Patterson, 1988, Skwarecki, 1988). The effects of such student controlled feedback are discussed by Corbett and Anderson (1988): It appeared that students do rethink and revise code when given the opportunity that is not afforded to them by the immediate feedback tutor. To be able to use more flexible tutoring strategies, the efficiency of modelling in the LISP-Tutor can be improved by running ahead of time in executing the production system; then, a trace of the problem solution is stored that retains whatever information is relevant to tutoring. This problem compilation enables functional modifications in the student model at relative low costs.

Comparing the Technical Specifications

Both systems are running on a DEC-VAX, and they are both written in LISP. PROUST is written in the, quite rare, LISP dialect T; the LISP-Tutor is written in the more widespread dialect Franz LISP. Noteworthy, whereas the LISP-Tutor is written in Franz LISP, it teaches Common LISP.

Table 2. Technical Specifications of PROUST and the LISP-Tutor

	PROUST	LISP-TUTOR
Computer System	DEC-Vax	DEC-Vax
Language	T-LISP	Franz LISP
Source Code	ca. 15.000 lines	ca. 10.000 lines
Development Time	8000 hours	6000 hours
Bug Identification	75%	45-80%

As may further be seen from Table 2, both systems are really big programs with development times of four to five years. In PROUST, approximately 75% of the program bugs is identified correctly. This includes multiple bugs; however, the system is able to handle only a few problems. In the LISP-Tutor, 80% of the bugs is identified correctly for the first lessons; this percentage drops to 45% for later lessons that include more difficult problems. These percentages for bug identification include only single bugs because multiple bugs are not allowed; however, in contrast to PROUST, the LISP-Tutor is able to handle a wide variety of problems.

7. RECENT DEVELOPMENTS

Several new systems, that are currently under development, are inspired by the architecture of Proust, the LISP-Tutor, or both. For instance, BRIDGE (Bonar et al., 1987; Bonar & Cunningham, 1988; Bonar & Liffick, 1988) is a system that is inspired by PROUST and which, like PROUST, uses goals, plans, and, eventually, actual code in its program diagnosis. However, instead of attempting to reconstruct a student's entire reasoning from the actual program code, BRIDGE forces students to prepare intermediate solutions. In a first phase, the solution must be expressed in--highly constrained--natural language; this corresponds to the goal level. In a second phase, the solution must be expressed in visualized plans; this corresponds to the plan level. Only in the third and last phase, the solution is presented in the actual programming code. Thus, instead of recreating the whole goal-plan problem solving trace, as done in PROUST, BRIDGE is tracking the trace one level at a time, focussing on the critical difficulties of that level. In assisting the student in each of the three phases, BRIDGE is using a graphically oriented interface that is far more flexible than PROUST's overly simple interface.

Another PROUST-inspired system that is under development is CHIRON (Sack, 1988). The diagnostic experts of PROUST as well as the LISP-Tutor locate bugs as the result of specific deviations from the correct code; plan- differences are used in PROUST and deviations from the ideal solution path are used in the LISP-Tutor. Weaknesses of this method include, among others, the facts that (a) plausible and implausible bugs are

handled equally, (b) sets of highly interrelated bugs, or 'super-bugs', cannot be easily handled, and (c) program parts are assumed to be independent. In CHIRON, the diagnostic expert will use a different approach that is called 'finding errors by overlooking them'. The method is geared toward the identification of high frequency programming errors; super-bugs are handled by automatically generating many similar, but slightly different patterns, and bugs are not corrected until all of the plans have been analyzed, that is, the diagnostic expert looks through the complete program before it makes corrections.

Based on the architecture of the LISP-Tutor, at CMU a tutor is developed for novice Pascal programming (Skwarecki, 1988). Another recent development that is inspired by the LISP-Tutor is the CMU-work on a generalized tutor that offers a framework to tutor an introduction to any programming language (Anderson & Skwarecki, 1986). To model the initial declarative knowledge PUPS structures are used (PenUltimate Production System; Anderson et al., 1988). These are basically schema-like structures, which prove critical to interpretive application of weak productions in problem solving: As a result of this activity, the knowledge is converted into use-specific productions. PUPS is intended as an implementation of the theoretical successor to the ACT* theory for skill acquisition (Anderson & Thompson, 1988); it is seen as the theoretical basis for the development of a new generation of ITS's.

GIL may serve as another example of a system currently under development at the Princeton University, which is heavily inspired by the LISP-Tutor (Graphical Instruction in LISP; Reiser, 1987; Reiser et al., 1988; Reiser, Friedmann, Kimberg, & Ranney, 1988). In the LISP-Tutor, the teaching expert may draw knowledge about programming from the domain expert; however, this domain expert always knows <u>what</u> action to take at any point in the solution, but not <u>why</u> that particular action would be effective. Thus, explanations to the student must always be prepared in advance for each situation in which feedback might be required. In GIL, however, the domain expert is designed to make the causal knowledge about programming explicit, so that the teaching expert can dynamically compute explanations from the problem solving rules in the domain expert and give more support to the planning phase. In addition, the GIL-Tutor is embedded in

a graphical programming environment, in which students build a program by connecting together objects representing program constructs into a graph. Many other proposals for intelligent programming tutors were presented at the Conference on Intelligent Tutoring Systems in Montreal (ITS '88).

8. DISCUSSION

After more than a decade of research and developments in ITS's for teaching programming, PROUST and the LISP-Tutor may be seen as two recent, well-known, and influential exponents of such systems. Moreover, as the systems are no longer under substantial development, it is the right time to draw up an inventory of the current state in intelligent programming tutors. In our view, systems like PROUST and the LISP-Tutor can then be best described, evaluated, and compared in terms of a generalized functional architecture for ITS.

In our evaluation and comparison, several conspicuous differences between the two systems were highlighted. For instance, whereas PROUST's domain expert is using the principle of goal-plan analysis to construct possible plan compositions that form a powerful basis for comparison with actual programs and the detection of multiple bugs, the LISP-Tutor uses a production system of an ideal student to construct solution paths that are a good basis for generating immediate feedback on bugs, but that are less valuable to diagnose multiple bugs. As another instance, it became clear that PROUST does not possess any kind of student model, whereas the LISP-Tutor offers a nice approach to building strong, qualitative student models.

Summarizing, it is clearly impossible to declare one of the two systems as the 'best' system; they both have their strengths and weaknesses. Actually, we feel that the primary importance of comparing ITS's is really not in pointing out a superior system, but in generating ideas to exchange weaknesses of one system with the strengths of other systems. As a matter of fact, several systems for teaching programming that are still under development are heavily inspired by either PROUST or the LISP-Tutor; however, they use a variety of strong ideas from other systems.

In the DISCS research group, we just started with a preliminary design-sketch for an ITS for teaching programming. First of all, we will not adhere to Pascal or LISP as subject matter; in our view, neither Pascal nor LISP will be extensively used in educational settings in the future. Our primary interest is in designing an ITS for tutoring the database query language SQL. Other interests include purely functional programming languages, such as MIRANDA (Turner, 1986) or TWENTEL, a functional language that is designed at the University of Twente (Van Der Hoeven, 1984).

More in particular, our preliminary design is constrained by three requirements. First, we want the students to have reasonable freedom to compose their own partial solutions, such as functions and procedures, and to construct complete programs out of those parts. As a consequence of this freedom, the identification of multiple bugs must be possible. The architecture of PROUST's or CHIRON's diagnostic expert may yield a starting point to reach this goal. As a second, and related requirement, we want to explicitly tutor on all phases in the design process, that is, on the goal, plan, as well as the code level. Forcing the students to prepare intermediate representations, as in BRIDGE, is thought to be desirable both from a pedagogical point of view and system efficiency, as the goal-plan problem solving trace can then be tracked one level at a time.

Finally, and probably most important, we want the possibility to vary instructional strategies as used by the teaching expert. The main reason is, of course, that our group is primarily interested in instruction so that the system must be suitable to conduct research on instructional design. We are aware of the fact that this is an ambitious goal; in most current ITS's for teaching programming the teaching expert is the least developed part. However, we feel that starting points may be offered by the LISP-Tutor's student model, which provides the teaching expert a good basis to adapt its instruction to individual students' needs, and GIL's domain expert, which offers the teaching expert a good basis to explain why a certain solution step is strategic or why a student step is in error.

AUTHORS' NOTE

We would like to thank Hans van Berne, Betsy van Dijk, Herman Koppelman, Hein Krammer, Albert Pilot, and Cees Terlouw for discussing programming tutors with us in STIPT.

REFERENCES

Adam, A., & Laurent, J.P. (1980). LAURA: A system to debug student programs. Artificial Intelligence, 15, 75-122.

Anderson, J.R. (1982). Acquisition of cognitive skill. Psychological Review, 89, 369-406.

Anderson, J.R. (1983). The Architecture of Cognition. Cambridge: Harvard University Press.

Anderson, J.R. (1986). Analysis of Student Performance with the LISP-Tutor. Pittsburgh: Carnegie Mellon University.

Anderson, J.R. (1987a). Skill acquisition: Compilation of weak-method problem solutions. Psychological Review, 94, 192-210.

Anderson, J.R. (1987b). Production systems, learning, and tutoring. In: D. Klahr, P. Langley, & R. Neches (Eds.), Production System Models of Learning and Development (pp. 437-459). Cambridge: MIT-Press.

Anderson, J.R., Boyle, C.F., Corbett, A., & Lewis, M. (1988). Cognitive modelling and intelligent tutoring. Artificial Intelligence, in press.

Anderson, J.R., Boyle, C.F., Farrell, R., & Reiser, B.J. (1987). Cognitive principles in the design of computer tutors. In: P. Morris (Ed.), Modelling Cognition (pp. 93-133). New York: Wiley and Sons.

Anderson, J.R., Boyle, C.F., & Reiser, B.J. (1985). Intelligent tutor systems. Science, 228, 456-462.

Anderson, J.R., Farrell, R., & Sauers, R. (1984). Learning to program in LISP. Cognitive Science, 8, 87-129.

Anderson, J.R., & Reiser, B.J. (1985). The LISP-Tutor. Byte, 10(4), 159- 175.

Anderson, J.R., & Skwarecki, E. (1986). The automated tutoring of introductory programming. Communications of the ACM, 29, 842-849.

Anderson, J.R., & Thompson, R. (1988). Use of analogy in a production system architecture. In: Vosniadov & Ortory (Eds.), Similarity and Analogical Reasoning. Cambridge: Cambridge University Press.

Barr, A., Beard, M., & Atkinson, R.C. (1975). A rationale and description of a CAI program to teach the BASIC programming language. Instructional Science, 4, 1-31.

Barr, A., Beard, M., & Atkinson, R.C. (1976). The computer as a tutorial laboratory: The Stanford BIP project. International Journal of Man- Machine Studies, 8, 567-596.

Bonar, J.G., & Cunningham, R. (1988). Intelligent tutoring with intermediate representations. Proceedings of the Conference on Intelligent Tutoring Systems 1988, Montreal: University of Montreal.

Bonar, J.G., Cunningham, R., Beatty, P., & Riggs, P. (1987). BRIDGE: Intelligent tutoring with intermediate representations (Tech. Rep. LRDC). Pittsburgh: University of Pittsburgh.

Bonar, J.G., & Liffick, B. (1988). A visual programming language for novices. In: Shi-Kuo Chang (Ed.), Visual Languages and Visual Programming, New York: Prentice Hall.

Cerri, S.A., Colombini, C., Grillo, M., & Mallozzi, R. (1985). RADAR: Reasoning on ADA rubbish. In: T. O'Shea (Ed.), Advances in Artificial Intelligence (pp. 199-208), New York: Elsevier North Holland Publishers.

Corbett, A.T., & Anderson, J.R. (1988). The LISP intelligent tutoring system: Research in skill acquisition. In: J. Larkin, R. Chabay, & C. Scheftic (Eds.), Computer Assisted Instruction and Intelligent Tutoring Systems: Establishing Communications and Collaboration. Hillsdale: Erlbaum Associates.

Corbett, A.T., Anderson, J.R., & Patterson, E.J. (1988). Problem compilation and tutoring flexibility in the LISP-Tutor. Proceedings of the Conference on Intelligent Tutoring Systems 1988, Montreal: University of Montreal.

Dijkstra, S., Krammer, H.P.M., Maaswinkel, R.M., Van Merriënboer, J.J.G., Van den Berg, K.G., Van Dijk, E.M.A.G., & Duijvestijn, A.J.W. (1988). DISCS: Designing instructional systems for computer science. Enschede, The Netherlands: University of Twente.

Hasemer, T. (1985). MacSolo/AURAC: A programming environment for novices. In: T. O'Shea (Ed.), Advances in Artificial Intelligence (pp. 189-198), New York: Elsevier North Holland Publishers..

Johnson, W.L. (1985). Intention-based Diagnosis of Programming Errors in Novice Programs (Tech. Rep. No. 246), Ph.D. Thesis. New Haven: Yale University, Department of Computer Science.

Johnson, W.L. (1986). Intention-based Diagnosis of Errors in Novice Programs. Paolo Alto: M. Kaufman. Johnson, W.L., & Soloway, E. (1985a). PROUST: Knowledge based program understanding. IEEE Transactions on Software Engineering, 11, 267-275.

Johnson. W.L., & Soloway, E. (1985b). PROUST: An automatic debugger for Pascal programs. Byte, 10(4), 179-190.

Johnson, W.L., & Soloway, E. (1985c). Design for Micro-PROUST. New Haven: Yale University, Department of Computer Science.

Lukey, F.J. (1980). Understanding and debugging programs. International Journal of Man-Machine Studies, 12, 189-202.

Miller, M. (1982). A structured planning and debugging environment for elementary programming. In D. Sleeman & J.S. Brown (Eds.), Intelligent Tutoring Systems, New York: Academic Press.

Murray, W.R. (1987). Automatic program debugging for intelligent tutoring systems. Computational Intelligence, 3, 1-16.

Pirolli, P.L. (1985). Problem Solving by Analogy and Skill Acquisition in the Domain of Programming. Phd. Dissertation, Pittsburgh: Carnegie-Mellon University.

Pirolli, P.L. (1986). A cognitive model and computer tutor for programming recursion. Human-Computer Interaction, 2, 319-355.

Pirolli, P.L., & Anderson, J.R. (1985). The role of learning from examples in the acquisition of recursive programming skills. Canadian Journal of Psychology, 39, 240-272.

Reiser, B.J. (1987). Causal models in programming. Proceedings of the Third International Conference on Artificial Intelligence and Education. Pittsburgh: University of Pittsburgh.

Reiser, B.J., Anderson, J.R., & Farrell, R. (1985). Dynamic student modelling in an intelligent tutor for LISP programming. Proceedings of the 9th IJCAI 1985, Los Angeles.

Reiser, B.J., Friedmann, P., Kimberg, D., & Ranney, M. (1988). Constructing explanations from problem solving rules to guide the planning of programs. Proceedings of the Conference on Intelligent Tutoring Systems (pp. 222- 229), Montreal, University of Montreal.

Reiser, B.J., Friedmann, P., Gevins, J., Kimberg, D., Ranney, M., & Romero, A. (1988). A graphical programming language interface for an intelligent LISP tutor. Proceeding of the Conference on Human Factors in Computing Systems (CHI 88), Washington DC.

Ruth, G. (1976). Intelligent program analysis. Artificial Intelligence, 7, 65-85.

Sack, W. (1988). Finding errors by overlooking them. Proceedings of the Conference on Intelligent Tutoring Systems, Montreal: University of Montreal.

Sack, W., Littman, D., Spohrer, J., Fertig, S., Hughes, L., & Soloway, E. (1988). From PROUST to CHIRON: Intermediate results are important. Proceedings of the Invited Workshop on Computer Based Learning Environments, Pittsburgh: Carnegy Mellon University.

Sack, W., Littman, D., Spohrer, J., Lilies, A., Fertig, S., Hughes, L., Johnson, W.L., & Soloway, E. (1986). Empirical Evaluation of the Educational Effectiveness of PROUST (Tech. Rep. No. 55). New Haven: Yale University, Department of Computer Science.

Singer, A., Ledgard, H., & Hueras, J. (1981). The annotated assistant: A step towards human engineering. IEEE Transactions on Software Engineering, 3, 353-374.

Skwarecki, E.J. (1988). Improving the engineering of model-tracing. Proceedings of the Conference on Intelligent Tutoring Systems (pp. 215- 221), Montreal, University of Montreal.

Soloway, E., Woolf, B., Barth, P., & Rubin, E. (1981). MENO-II: Catching run-time errors in novice's Pascal programs. Proceedings of the IJCAI- 1981, Vancouver.

Soloway, E., Rubin, E., Woolf, B., & Bonar, J. (1984). MENO-II: An AI-based programming tutor. Journal of Computer-Based Instruction, 10, 20-34.

Stijnen, J., & Offereins, M. (1988). Analyse van de CMU LISP-Tutor [Analysis of the CMU LISP-Tutor]. Enschede, The Netherlands: University of Twente, Department of Computer Science.

Turner, D. (1986). An overview of MIRANDA. SIGPLAN Notices, 21, 158-166.

Van Der Hoeven, G.H. (1984). Preliminary Report on the Language TWENTEL (Tech. Rep. No. INF-84-5). Enschede, The Netherlands: University of Twente, Department of Computer Science.

Van Merriënboer, J.J.G. (1987). Intelligente tutorsystemen voor het leren programmeren in een hogere computertaal: Verslag van een studiereis naar de VS [Intelligent tutoring systems for learning to program in high level languages: Report on a study trip to the USA]. Enschede, The Netherlands: University of Twente, Department of Instructional Technology.

Van Merriënboer, J.J.G. (1988). Strategies for programming instruction in high school: Program completion vs. program generation. Manuscript submitted for publication.

Van Merriënboer, J.J.G. (in press). Relationship between cognitive learning style and achievement in an introductory computer programming course. Journal of Research on Computing in Education.

Van Merriënboer, J.J.G., & Krammer, H.P.M. (1987). Instructional strategies and tactics for the design of introductory computer programming courses in high school. Instructional Science, 16, 251-285.

Wallach, B. (1987). Development strategies for ICAI on small computers. In G. Kearsly (Ed.), Artificial Intelligence and Instruction (pp. 305-322). Reading, MA: Addison-Wesley.

Wertz, H. (1982). Stereotyped program debugging: An aid for novice programmers. International Journal of Man-Machine Studies, 16, 379-392.

SOME ISSUES ON THE CONSTRUCTION OF COOPERATIVE ITS

Gellof Kanselaar, Jerry E.B. Andriessen, Yvonne F. Barnard & Gijsbert Erkens

University of Utrecht

Abstract

In this article we will discuss the cooperation paradigm for computer assisted learning. In Intelligent Cooperative Systems (ICS) student and system work together as partners on problem solving tasks. The requirements for ICS are: complementary abilities or information, mixed control, mixed initiative and common goal between student and system. The advantages of the cooperative approach as a learning principle are discussed. Some implications for the representation and implementation of the interaction component in ICS are examined. Examples from two research projects are given that are concerned with communicative actions and the framework to interpret these actions.

1. INTRODUCTION

In a recent chapter on cognitive principles in the design of computer tutors Anderson et al. (1987) state: "The intelligent tutoring paradigm states that the most promising way for bringing intelligence to bear in delivering instruction is to emulate a private human tutor" (p. 93). "Students appear to learn information more effectively if they are presented with that information during problem-solving rather than during instruction apart from the problem-solving context" (p. 105).

The concurrence and coordination of information exchange during the process of problem solving seems to be a crucial aspect. One could expect the effectiveness of intelligent tutoring to increase if both student and tutor are allowed to communicate about each other's knowledge about the problem and their respective problem-solving procedures. In cooperative

settings, however, this kind of tuning is a natural requirement.

In this article we will discuss the cooperation paradigm for computer assisted learning. In this paradigm the attention shifts from the problem of domain knowledge to the problem of communication. Some issues are discussed on the advantages of such an approach and on the directions that future research on intelligent cooperation could take. First, the cooperative approach is defined within the field of intelligent tutoring and its characteristics and advantages will be described. Then, we will raise some issues that deserve attention from researchers. Thirdly, we will present some relevant examples from our own research. Finally, some implications for the construction of cooperative systems will be put forward.

2. APPROACHES IN INTELLIGENT COMPUTER ASSISTED LEARNING

In research on intelligent computer assisted learning, two main approaches can be distinguished.

Tutoring systems

One approach is the development of tutoring systems acting as a teacher who guides the student and controls the learning path of the student. The system determines which problems the student has to solve, when a correct solution has been reached and when it is necessary to provide guidance. The control lies mainly with the system. Emphasis lies on the development of domain expert- and diagnosis-modules, and the effectiveness of such tutoring ultimately depends on instructional and curriculum expertise in specific knowledge domains.

Open learning environments

The other approach is the development of open learning environments in which the student is able to take over control and to determine his/her own learning path. These systems can take the shape of a simulation environment, a laboratory, a microworld etc. Control lies mainly with the student. The goal of many of these computer-based learning environments typically is to encourage learning without specifying it directly (Levin & Waugh, 1988). The effectiveness of this

approach depends on the validity of a specific learning principle, that of (eventually guided) discovery learning.

In general, one could say that in both approaches "who is in command" is not a matter of negotiation but is predetermined. Control over the learning path lies either with the student or with the system. Who has control is not dependent on the ongoing problem solving process nor on the ongoing interaction process.

3. INTELLIGENT COOPERATIVE SYSTEMS

Presently, however, a third approach seems to emerge: systems that are meant to cooperate with the student. We propose to call these systems Intelligent Cooperative Systems (ICS). In this approach neither the student nor the system has complete control of the learning path. They both work together as "equal partners" on a problem solving task.

At this moment we do not know of systems that are fully cooperative. Yet more and more research within the field of ITS aims at a more cooperative approach. Examples are: mixed initiative systems, systems where the student acts as a teacher to the system, help-systems (Anzai, 1987; Law and Ki, 1987; Scheftic and Wood, 1987, Breuker, Winkels and Sandberg, 1987).

What should be the characteristics of ICS?

We could define a cooperative learning situation as one in which two or more students work together to fulfill an assigned task within a particular domain of learning in order to achieve a joint product. From this definition we can infer the following criteria for ICS.

Complementary abilities or information
Cooperation is not very useful when one is able to complete the task by oneself. Only when the participants have abilities or information that are complementary, cooperation can be fruitful. ICS requires tasks that call for cooperation to reach a successful completion. This could also imply that an ICS does not have complete knowledge about a domain and is not able to solve all the problems encountered.

Mixed control
In cooperative learning situations none of the participants is able to determine the process one-sided. The participants are dependent on each other's cooperation. System and student in ICS should interact with each other, give and ask information, discuss inferences being made and strive for mutual agreement. Both participants have the opportunity to take control of the exchange and processing of information in the interaction.

Mixed initiative
Both system and student have to be able to take the initiative in interaction. They must be able to take initiative in asking questions, in making remarks, in transferring information, in suggesting solutions etc.

Common interest and common goal
In cooperation system and student must have a common interest in solving the problem at hand. They have to reach common goals and common sub-goals that determine the flow of the problem solving process.

Why could this approach be useful?

Cooperative learning
In recent educational research there is a great interest in cooperative learning. Under certain conditions learning in problem solving domains appears to be more productive in peer-cooperative settings than in individual or teacher-pupil interactional settings (Webb, 1982). The causes for favorable outcomes of cooperative learning experiments are not yet fully understood. Some explanations point at findings that pupils profit from the necessity to coordinate their actions and that pupils can understand each other's line of reasoning better. Other researchers contribute the positive effects to situations in cooperative learning that evoke a so-called socio-cognitive conflict (Doise, Mugny and Perret- Clermont, 1975). This conflict obliges the pupil to restructure his or her cognitive representation when confronted with contradictory information of the partner. However, most educational researchers agree that cooperative learning, under the right conditions, can be a valuable way of learning problem solving abilities.

Reflection
Cooperation is not just working together pleasantly. Cooperation can be a hard struggle. One has to defend one's own standpoint, give arguments, challenge information transferred by the partner etc. Cooperation forces the partners to verbalize and defend their choices in the problem solving process. These activities enhance reflection on one's own thinking process.

Forcing a student to verbalize during problem solving can be a powerful instrument for the system to get a view on the internal cognitive processes of the student. A system should benefit from this "naturally" begotten data about the cognitive processes to construct its student-model.

Open domains
For some learning domains it will remain impossible for an ITS to have a complete domain-representation. In open domains (for example story writing) it is not feasible to determine the learning process completely. In these domains an approach can be fruitful in which the system seeks for content information from the student and tries to give suggestions based on its own (limited) knowledge.

Imagine an ICS for story writing. The system contains schematic and strategic knowledge about how to write a story. This system could have an extended knowledge of writing processes such as: planning, organizing content, formulating and revision. However, the system need not to possess any content-knowledge at all. This last kind of knowledge is in the possession of the student. She or he knows what the story is about, but has to cooperate with the system in the process of writing the story down and evaluating the written product.

4. MODEL OF COOPERATIVE PROBLEM SOLVING

At the University of Utrecht research is being conducted within the field of cooperative learning and the development of intelligent educational computer systems. One project, the DSA-project (Erkens & Barnard, 1988), is concerned with research on the relation between information processing and information exchange during cooperative problem solving. In this project a simulation-program has been constructed that

models two pupils cooperating on a problem solving task. The model advanced in this project (Figure 1) may serve as a prototype for further development of cooperative computer-programs.

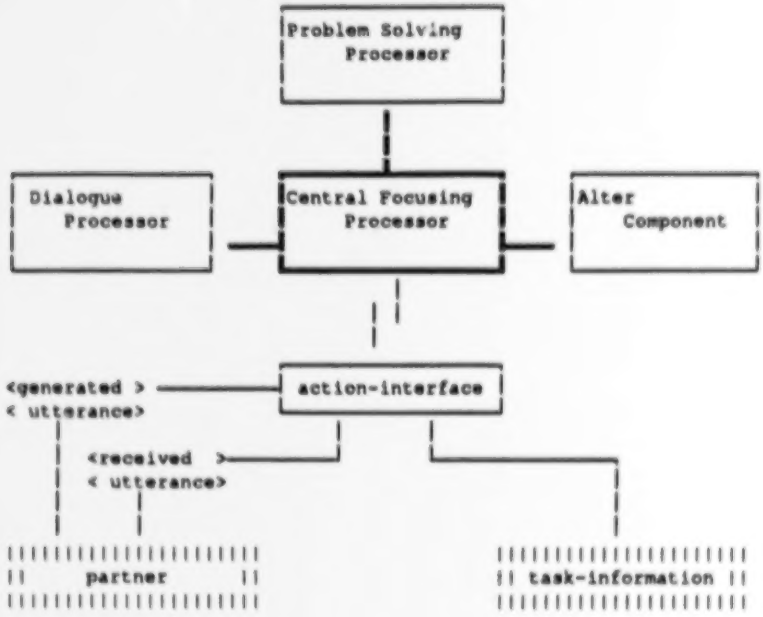


Figure 1: Model of problem solving and dialogue processing

In the DSA-model the following components can be distinguished:

- problem solving processor: contains knowledge about the domain and problem solving strategies

- dialogue processor:	contains knowledge about interaction processes, strategies for analyzing and generating communicative actions
- partner model:	contains a belief-system about the communicative and problem solving activities of the partner
- central focusing processor:	this component controls the flow of information between the different components and combines the results from the processes that go on in the other components. This component contains the cooperation strategies.
- interface:	translates the input from the partner in an internal representation and vice versa.

In ICS it is crucial that the possibilities of interaction with the student are extensive and flexible. Therefore we will focus on the aspect of interaction.

5. STUDENT-SYSTEM INTERACTION

To reach a solution student and system are engaged in a two-way process of communication. The crucial notion in the analysis of communication is Grice's principle of cooperation: in order for communication to be efficient and effective participants serve the common goal of making each utterance relevant, informative, true and clear (Grice 1975). One consequence of this principle is that in natural conversation, without evidence to the contrary, every utterance is interpreted as coherent within the context of earlier utterances.

Ideally, student and ICS are partners in a conversation. Both possess certain knowledge and both need information of

the other in order to solve a problem. What is essential here is that both have the common goal of solving a problem and for this it is necessary that both have a common understanding of the nature of this problem. The path to a solution is a matter of debate. In trying to understand the student's method, the system needs to interpret the student's actions. The system must be able to communicate with the student, in order to comprehend the student's actions more clearly, or to correct the student's actions.

We therefore in ICS need a framework for the interpretation and elicitation of different types of task-directed communicative actions. This requires results from empirical research on cooperation in problem solving contexts. We need to study the process of information exchange during cooperative problem solving to know what communicative actions are performed for what purposes during task execution. Furthermore, we need to develop criteria for the interpretation of communicative actions according to a coherent problem representation.

In the next paragraphs we will discuss these two central aspects: characteristics of communicative actions during cooperative problem solving, and the interpretation of those communicative actions related to propositional content. We will illustrate these two issues with empirical data from two research projects carried out at our institute.

6. ILLUSTRATIONS FROM TWO RESEARCH-PROJECTS

We will base our considerations mainly on the results of two projects, one is the already mentioned DSA-project, the second is a project on computer assisted text production. In this second project one of the aims is to discover some principles behind coherent story construction. Both projects involve cooperative problem solving tasks with 10 to 12 year old pupils.

Communicative actions: the DSA-project

The DSA-project is aimed at gaining insight in the relationship between the cognitive aspects of information processing and the communicative process of information

exchange during cooperative problem solving. These processes are modeled by using the method of protocol-analysis and computer-simulation. Dialogues between pupils cooperating on a problem solving task, are analyzed and simulated. The simulation-program will be the core of an ICS that interacts with a pupil in cooperative problem solving.

Example from a dialogue-protocol
First we would like to present an example, a few lines from a protocol. The cooperation-task the pupils work on is the solving of a logical problem. They have to deduce several characteristics of some children described in the task-material. Both pupils have different information in the form of two letters, supposed to come from two of the children mentioned in the task.

In the example the pupils are trying to find out which child does gymnastics and who sleeps in which tent in a summer camp. Pupil A has information in a letter from a boy named Peter. Both pupils know that the red tent stands between the blue and the yellow tent.

1	A:	And Chris does gymnastics
2	B:	Oh,..yes
3	A:	(reads from the letter) "John is the boy who sleeps in the red tent next to me"
4	B:	Who,..eh, who says so? Chris?
5	A:	Eh, yes, Chris does gymnastics
6	B:	Then Chris sleeps in blue or yellow,
7		...if he sleeps next to John
8		He sleeps in blue or yellow
9	A:	I don't know if he sleeps next to John

When we look at the fragment, in line 4 we see the first problem of coordination. Pupil B does not understand who is the "me" mentioned in line 3. He confuses Peter, the writer of the letter, with the already mentioned Chris. Because pupil A misunderstands the question and repeats that Chris does gymnastics, pupil B concludes in line 6 that Chris sleeps next to John. In line 8 we see that pupil B repeats his conclusion, this kind of re-concluding has an important function for the coordination between partners. In line 9 we see that pupil A

also actively engages in the process of coordination, he does not accept the solution gratuitous, but checks it.

Coordination

In the example from a protocol we already noticed the role of coordination. One of the main findings of our analyses of the protocols, is that pupils spend a great deal of their time in these task-oriented dialogues on controlling and coordinating activities such as checking of plausibility and giving information about the status of information transferred.

Checking procedures were found to play an important role in the coordination of actions in the analyzed task-dialogues. In the dialogues checking can be found in the form of:

- check questions, for example: are you sure?
- confirmations
- denials
- acceptations
- counter-arguments

Pupils often give an explicit indication of the status of information transferred. Examples are: "I think", "I know for sure", "I don't know". On the average 25% of all utterances in the protocols have such indications.

We distinguish two kinds of checking, checking of the plausibility and checking on contradictory information.

* Plausibility checks
3-7.5% of all utterances in the protocols are plausibility-checks.
Examples of plausibility-checks are:
A: "Peter sleeps in the white tent"
B: "Is that so?" or B: "Yes?"

* Contra-checks
Likewise, 3-7.5% of all utterances in the protocols are contra-checks.
Examples of contra-checks are:
A: "Peter sleeps in the white tent"
B: "No, in the brown one." or B: "Can't be."

In these protocols checking takes a substantial part of all the utterances made during task-performance. We think that before information can be incorporated in a cognitive

representation, this checking procedures on relevance and coherence with earlier knowledge is necessary. People can only relate external information to their own knowledge if it is perceived as relevant, plausible and consistent (see also Ausubel and Robinson, 1969).

Facilitating cognitive representation
Pupils cooperating give redundant information, context-indications, recapitulations etc. These serve also an important function for the receiver to distinguish the kernel of the information transferred and facilitates the integration of this information into the cognitive representation of the problem at hand.

When we look at sequences of communicative actions that occur frequently in the protocols we can distinguish some prevailing dialogue-patterns. The basic pattern is the following:

1. Asking for attention	A: "Eh, look"
2. Transferring new information	A: "Peter sleeps in white"
3. Accepting or confirming	B: "Yes"
4. Re-concluding	A: "So, white"
5. Writing down the solution	A & B: writing

Coordination occurs by means of signaling, asking for attention before new information is transferred, and by means of explicitly accepting and re-concluding before writing down the solution. Partners make sure by these means that they both share the same understanding of the problem.

Interpretation framework; the computer assisted text production project

We will discuss the second central aspect, how to interpret communicative actions and how to evaluate their relevancy and coherency, by presenting illustrations from a second project, the Computer Assisted Text Production project (Andriessen, 1988). In the project pupils have to compose a story by choosing a sentence from a set of four possible alternatives. These alternatives are presented on screen, the pupils have to type in the number of the sentence of their choice. The

sentences contain slots in which the names of the main characters or pronouns referring to them have to be filled in. The chosen sentences have to be added to the story and the resulting text is constantly shown on the screen. In this way a story of 20 lines is constructed. A computer-program is being developed that tries to understand stories during the process of their composition. Sentences that are added to the story are evaluated for coherency. The program builds up a model of the story sentence by sentence, using an interpretation-framework. The relevancy of this research to ICS concerns the construction of an interpretation-framework as a means to evaluate communicative information.

Example from story construction protocol
First, we will give an example of a story-protocol. Consider the following passage:

[example 1]
1: John sat at the back of Charles' bicycle.
2: You have a flat tyre!, John said.
3: A flat tyre, that is not possible, Charles said.
4: Well, look for yourself!, John said.
5: How can that be?, Charles said.

At this moment, the following alternatives are presented:

a: # thought about it for a moment.
b: Maybe, because someone sat at the back, said #.
c: A hole in the inner tube, # said.
d: I could have hit a stone, said #.

This story starts with two children having a problem. The first four sentences are concerned with the introduction and confirmation of this problem: a flat tyre. Sentence 5 is an inquiry into the cause of this problem. There are several possible continuations, but there are quite strict constraints. The next sentence should react to the question. It should be a relevant reaction, by the right person.

At first glance it might seem that the four alternatives each are a probable candidate for the next sentence. Closer reflection might reveal that in this context the person sitting at the back of a bicycle is not likely to state that

an anonymous "someone" is the cause of a flat tyre (alternative b).

With a different first sentence (example 2), alternative b) is a more plausible reaction, because this first sentence creates a different context of interpretation: "Two children at the bicycle-shelter".

[example 2]
1: John met Charles at the bicycle-shelter.
2: You have a flat tyre!, John said.
3: A flat tyre, that is not possible, Charles said.
4: Well, look for yourself!, John said.
5: How can that be?, Charles said.

With a different fifth sentence (example 3), only alternative a) is a coherent continuation. The other alternatives are concerned with giving a causal explanation and do not answer the question "What shall we do now?".

[example 3]
1: John sat at the back of Charles' bicycle.
2: You have a flat tyre!, John said.
3: A flat tyre, that is not possible, Charles said.
4: Well, look for yourself!, John said.
5: What shall we do now?, Charles asked.

The computer-program has to interpret each incoming sentence as a reader would do. Therefore we regard the program as a simulation-model of the reader. The reader-model builds up a frame-based representation of the developing story and has to integrate each new sentence into this representation. The frames contain essential information about the story content, and the plans, goals, actions, objects and moods of the story-characters. When a new sentence is added to the story the reader-model evaluates the appropriateness of that sentence on the basis of a production-system of coherency-rules.

The example in figure 2 contains an excerpt from the story representation. Here, it serves as a conceivable framework for the evaluation of story coherence.

In cooperative problem solving, a similar model could function as an interpretation framework for incoming

STORY REPRESENTATION

person 1		person 2	
name:	John	name:	Charles
location:	bicycle shelter	location:	bicycle shelter
objects:	-	objects:	bicycle
problem:	-	problem:	flat tyre
goal:	-	goal:	know cause

current theme: the flat tyre of Charles' bicycle

story focus: Charles wants to know the cause

coherency constraints:

conversational:	answer Charles' question
rhetorical:	discuss cause or think about it
theme:	current (=the flat tyre)

Figure 2: Story representation

communicative information. Interaction with an ICS could be based on such representations at the content-level. Such interpretation frameworks can function at the level of the domain representation as well as in the modeling of a partner. For a specific domain, specific coherency constraints on the content-level should be derived.

The problem of parallel representation

One of the main problems in interaction is the so-called problem of parallel representation. In general, this means that pupil and system, or two pupils, do not have the same understanding of the problem. They interpret each others problem representation according to different standards. Detection of this kind of discrepancies need not to occur immediately. Misunderstandings can persevere a long time before they are discovered.

We will give an example (see figure 3) from the story construction project to illustrate the effects of parallel representation.

Story produced on screen

Yasmin sat at the back of Miriam's bicycle.
I have to go to my guitar-class, Yasmin said.
That's not possible, Yasmin said, a flat tyre.
But that tyre is completely flat, Yasmin shouted.
What shall we do now? Yasmin asked.
Miriam thought for a while.
Because there was a nail on the road, Yasmin said.

Additional sentences orally produced

-> I'll take you there, Miriam said.
->
No problem, I'll have it fixed in 15 minutes.

-> No problem, Miriam said.
-> We'll just inflate it, Miriam said.
->
I know how it came about!, Yasmin said.

Figure 3: Example of parallel representation of system and pupil in story construction

In the example in figure 3 we see the construction of a story. In the left-hand column we see the actual story as it was produced by the pupil and was displayed on screen. In this example an extra instruction was added to the task: the pupil had to conceive a following sentence each time before the alternatives were presented and had to say it aloud to the experimenter. In the figure these sentences are the arrow headed (->) sentences in the right-hand column. The instruction stated explicitly that the self-generated sentences were not included in the story, and these sentences were not presented on the screen (the child only could view the sentences on the left).

However, the child composed a story based on a representation that included both chosen and self-generated sentences. The example shows that the actual story can only be adequately understood as a coherent dialogue when also the additional sentences (in the right-hand column) are taken into account. The computer-program, however, only knows about the

sentences chosen from the alternatives, and thus has a rather different story representation as the pupil.

This example points at the problem of different representations. In order for student and system to understand each other, both should check regularly the extent to which they have a common problem representation. As was indicated in the section about the DSA-project, this kind of coordination and adaptation processes were found frequently in interactions between pupils. ICS should profit from these natural phenomena and stimulate these checking processes in order to control or to prevent the occurrence of different representations.

8. IMPLICATIONS FOR THE CONSTRUCTION OF THE INTERACTION COMPONENT IN ICS

Broadening the bandwidth

The empirical results of these two projects inform us about communicative actions that occur during cooperatively solving complex problems and on how to evaluate the appropriateness of these actions. If we want an ICS to give the student the possibility to interact with the system in the same manner, what are the consequences for implementation and representation?

The "intelligence" of computer assisted educational systems is a function of the degree to which the system is able to recognize the information processing of the student and to adapt itself to it. This depends on the "bandwidth" (VanLehn, 1988) of the system. We define the bandwidth of the interaction component as the variety of propositional and communicative information transferred by a student the system is able to receive and to process. The bandwidth sets the limits to the adaptability of the computer-program to the individual student. The interaction channel between system and student is the bottleneck through which the system receives the information necessary to model the student (see also Douglas, 1988). This applies not only to ICS but to all computer assisted educational systems.

In most ITS this information is extracted by means of an interpretative module on basis of the products and intermediate solutions the student transfers during ongoing

problem solving. A narrow bandwidth of interaction directly limits the depth and the validity of the student modeling that can be ideally reached. In other words, the less possibilities for interaction a student has, the more inferences the system has to make for the construction of a student-model. This will increase error probability. Vice versa, if the system has only limited interactive possibilities, no matter how sophisticated the student-model will be, differentiated adaptation to the individual student is hardly feasible.

Broadening the bandwidth of the interaction component is not identical to developing new interfacing-techniques such as menu-driven systems, natural language interfaces, speech recognition and synthesizing systems, auditory-video connections, graphical presentations etc. (Miller, 1988). If we plead for broadening the interaction, then we refer to the variety of communicative functions more than to all the different forms in which the functions may be put. As an illustration let us look at an example from natural discourse. In natural dialogues, the listener has direct control over the kind of information that will be transferred by the speaker by means of only two simple communicative actions: confirmatory head nodding and looking blank. In reaction to head nodding or confirming the speaker will continue the line of thought he or she is transferring. However, when the listener looks blank or frowns, the speaker will tend to elaborate on, explain or rephrase the information at focus. In most computer applications we do have the possibility to nod, for example by means of pressing the return-key. However, we highly miss the possibility of looking blank or raising our brows as the text scrolls before our eyes.

The construction of a partner-model in ICS

What differences in representation do we observe if we compare the construction of a partner-model in ICS and student-modeling in ITS? Without giving final answers, some preliminary remarks can be made on this topic:

1. How to interpret the communicative actions of the other? The partner-model in ICS has to contain also information about the procedures used at the communicative level. The ICS has to interpret the communicative actions of the other. This

interpretation must be driven by the expectations and demands derived from the information the partner transfers by his or her communicative actions.

2. How to restrict the partner-model to information that is useful?
We could answer this question in the same manner as Ohlsson (1986) does on student modeling in ITS. Ohlsson pleads for a direct relation between student- model construction and the didactic action repertoire an ITS possesses. Likewise, we could relate the partner-model to the cooperative goals and communicative action repertoire an ICS should possess.

3. How deep should the partner-model be?
To put this question in another way, what has to happen if the ICS encounters a line of thinking of the student it cannot recognize or integrate with its own representation? To answer this question, let us take a look at what happens if two cooperating pupils do not understand each other? They will explain and rephrase their arguments. They try to find a common representation in which their common goal can be fulfilled. We hope that the strength of a cooperation-approach for computer assisted learning will not only be in the issue of learner control, but also in the possibility for the system to oblige the learner, in a natural manner, to stay on grounds that the system can understand.

The representation of interactional and cooperational plans

One of the problems for the implementation of ICS will be the representation of plan-based structures that occur in cooperative dialogues. The maxim of mixed control and flexibility of interaction implies that these structures can hardly be represented on a rigid script-like basis. Plans need not always be completely executed.

When a plan has been interrupted, return to this plan is not always necessary or even wanted. We encountered a similar problem in a project on mathematical tutoring. When a problem at hand was not correctly solved, the program started a long remedial interaction. It proved to be ineffective or even confusing to return to the same problem when the remedying was

finished. So, sometimes it is better to start with a new topic or with a new plan.

9. CONCLUDING REMARKS

We do not think that the cooperative approach will be the panacea for all learning situations in all educational domains. Nevertheless, cooperation in itself may be an effective educational method for computer assisted learning. Particularly in those domains in which the learning of problem solving abilities is central. If we view cooperation as an educational method in itself, we do not contend that in ICS there will be no room for a didactic component, which contains teaching strategies and didactic principles. In fact, a cooperative system could be expanded with a didactic component containing instructional knowledge. The essence of the cooperative approach lies within the criteria of complementary abilities or information, mixed control, mixed initiative and common goal. On account of the first criterion, complementary abilities, it is feasible that the ICS has more didactic expertise than the student. However, in ICS it is the common problem solving state that will determine when and where the didactic action will be offered on the basis of a mutually agreed upon goal.

The construction of a working ICS that really cooperates like a fellow-student still has a long way to go. Study of task-oriented human interaction and developmental research on ICS construction should take us further along this way. The most attention should be devoted to the interaction component. This research should be conducted within domains of cooperative interaction during problem solving.

Research on ICS development has as its final goal the application of such a system in the classroom. In the near future this will not be a realistic option. The first benefit of this type of research will not be the construction a working application, but the gaining of more insight in the various aspects of cooperative problem solving and interaction processes. In this way ICS research could be an important contribution to the study of learning and education.

REFERENCES

Anderson, J.R., Boyle, C.F., Farrell, R., & Reiser, B. (1987). Cognitive principles in the design of computer tutors. In P. Morris (Ed), Modeling cognition. New York: Wiley.

Andriessen, J.E.B., (1988). The story-construction process: perspectives for intelligent tutoring. Paper presented at the AERA annual meeting, New Orleans.

Anzai, Y. (1987). Architecture for problem understanding in physics problem solving. Paper presented at the Third International Conference on Artificial Intelligence and Education, Pittsburgh.

Ausubel, D.P., & Robinson, F.G. (1969). School learning. New York: Holt, Rinehart & Winston.

Breuker, J.A., Winkels, R.G.F., & Sandberg, J.A.C. (1987). Coaching strategies for help-systems: EUROHELP. Paper presented at the Third International Conference on Artificial Intelligence and Education, Pittsburgh.

Douglas, S.A. (1988). Detecting and repairing tutoring failures. Manuscript CIS-TR- 88-09, Department of Computer and Information Science, University of Oregon.

Erkens, G., & Barnard, Y.F. (1988). Computer simulation of information processing and information exchange during cooperative learning. Paper presented at the AERA annual meeting, New Orleans.

Doise, W., Mugny, G., & Perret-Clermont, A.N. (1975) Social interaction and the development of cognitive operations. European Journal of Social Psychology, 5, 367-383.

Grice, H.P. (1975). Logic and conversation. In P. Cole and J.L. Morgan (Eds), Syntax and semantics, vol.3: Speech acts. New York: Seminar Press.

Law, N., & Ki, W.W. (1987). A.I. Programming environment as a knowledge elicitation & cognitive modeling tool. Paper presented at the Third International Conference on Artificial Intelligence and Education, Pittsburgh.

Levin, J.A., & Waugh, M. (1988). Educational simulations, tools, games, and micro- worlds: computer-based environments for learning. International Journal of Educational Research, 12, 1, 71-79.

Miller, J.R. (1988). The role of human-computer interaction in Intelligent Tutoring Systems.In M.C. Polson and J.J. Richardson (Eds), Foundations of Intelligent Tutoring Systems. Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Ohlsson, S. (1986). Some principles of intelligent tutoring. Instructional Science, 14, 293-326.

Scheftic, C., & Wood, G.D. (1987). Querious: a system students can teach. Paper presented at the Third International Conference on Artificial Intelligence and Education, Pittsburgh.

VanLehn, K. (1988). Student modeling. In M.C. Polson and J.J. Richardson (Eds), Foundations of Intelligent Tutoring Systems. Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Webb, N.M. (1982). Student Interaction and Learning in Small Groups. Review of Educational Research, 52, 421-445.

ARTIFICIAL INTELLIGENCE TOOLS FOR GRAMMAR AND SPELLING INSTRUCTION *

Fiany Pijls, Walter Daelemans & Gerard Kempen

University of Nijmegen
Free University of Brussels

Abstract
In The Netherlands, grammar teaching is an especially important subject in the curriculum of children aged 10-15 for several reasons. However, in spite of all attention and time invested, the results are poor. This article describes the problems and our attempt to overcome them by developing an intelligent computational instruction environment consisting of 1. a linguistic expert system, containing a module representing grammar and spelling rules and a number of modules to manipulate these rules, 2. a didactic module and 3. a student interface with special facilities for grammar and spelling. Three prototypes of the functionality are discussed: BOUWSTEEN and COGO, programs for constructing and analyzing Dutch sentences; TDTDT, a program for the conjugation of Dutch verbs.

1. GRAMMAR AND SPELLING INSTRUCTION: PROBLEMS

In The Netherlands and in Flanders, grammar teaching is an especially important subject in the curriculum of children aged 10-15. Three reasons are the following:

1. Grammatical knowledge is generally considered necessary to improve the quality of writing skills.
2. The orthography of Dutch depends to a considerable extent on grammatical relationships within the sentence. In many cases the correct spelling of a word cannot be derived from memorized word spellings or word-based phoneme-to-grapheme rules. A typical example is provided by the Dutch homophonous passive auxiliaries word and wordt. Choosing

between them presupposes the explicit application of subject-verb agreement rules.
3. In secondary schools at least three foreign languages are taught (French, English and German). A fair amount of grammatical knowledge is needed here, e.g. for determining case endings of German adjectives and nouns.

For these reasons, much attention and time is invested in grammar and spelling teaching. Nevertheless, the results are poor. In our opinion this is caused by, among other things:
1. the fact that grammar is presented as a set of rules for analyzing rather than generating sentences. This causes a low level of transfer from grammar instruction to writing skills. Many students are unable to apply the grammar and spelling rules they have been taught, when composing an essay;
2. uninformative feedback. After making errors, students very often just write down corrections given by the teacher, and do not attempt to find reasoned solutions;
3. dull and uninspiring exercises, which can be executed without much thinking on applicable rules;
4. lack of notational uniformity of current school grammars (e.g. several teachers in one school using different or even conflicting notations);
5. insufficient explicitness concerning the relations between grammatical functions (e.g. subject, direct object, predicate) and grammatical categories (phrases, constituents, parts of speech).

For a a detailed overview of the problems of grammar and spelling teaching in Dutch, see Van Dort-Slijper (1984) and Assink (1983).

2. GRAMMAR AND SPELLING INSTRUCTION: INFORMATION TECHNOLOGICAL TOOLS

Several attempts have been made to solve these shortcomings of grammar and spelling instruction by means of traditional Computer Aided Instruction. Although some problems can be solved in this way (immediate, consistent, and informative feedback is made possible, for instance), CAI cannot bring a

complete solution (Kempen, Schotel & Pijls (1985) evaluate the literature on CAI programs for grammar and spelling teaching in Dutch and other languages.) The major drawback of these programs is their lack of grammatical knowledge. This makes it impossible to produce a sufficiently detailed diagnosis of the learner's problems, and to come up with explanatory feedback upon errors made.

In a four-year research project which was started in September 1985, we attempt to overcome some of these limitations by developing an intelligent computational instruction environment for grammar and spelling. It consists of three main components:

1. A linguistic expert system, which is knowledgeable about the grammatical and orthographic subject matters, that are taught to the students. Subcomponents are, among other things, a sentence generator and a parser using a notation for syntactic structures which comes close to notations that are in use in popular school grammars but is sufficiently formalized. To this purpose we devised a simplified version of Kempen & Hoenkamp's (1984) Incremental Procedural Grammar (IPG). This grammar is accompanied by a (two-dimensional) tree notation for syntactic structures which is much more transparant than the one-dimensional notations currently in use in Dutch school grammars.
2. A didactic module including a subcomponent for diagnosing the learner's knowledge, a bug catalogue (a list of frequent mistakes to guide the diagnosis procedure), a tutoring module (knowledge about what knowledge to impart to a particular learner and how to do it), and an exercise generator.
3. A student interface. An important component of this interface is TREE DOCTOR (written by Desain, 1985), a powerful graphical tree editor. It can be used not only for displaying IPG parse trees, but also for manipulating them in various ways. TREE DOCTOR allows the student to modify syntactic trees (e.g. deleting and adding branches and nodes, attaching branches to other nodes, changing left-to-right order of nodes, etc.) according to the 'direct manipulation' style of user interfaces advocated by Shneiderman (1982). An example is given in Figure 1. It shows a student moving the attachment point from one node

to another, thus correcting an unintended parse tree of an ambiguous sentence. TREE DOCTOR also offers a facility for 'closing' trees, i.e. for hiding the details of a subtree into a triangle. By asking TREE DOCTOR to display certain parts of a tree in a closed or open manner, the teacher can 'zoom in' onto those aspects of the tree that are most relevant for a certain student or group of students.

Presently, the system is far from being fully implemented. However, we did finish three prototypes embodying part of the functionality that we are aiming at. One is BOUWSTEEN (building block), a program allowing students to compose sentences out of parts of speech. BOUWSTEEN runs on an IBM-AT. A second program, COGO (Schotel & Pijls, 1986), allows learners to practice sentence analysis on sentences they have composed themselves. A third prototype is TDTDT (TDTDT Diagnoses Trouble with DT; Daelemans, 1987), a small teaching environment for the conjugation of Dutch verbs. Both COGO and TDTDT are running on a Symbolics Lisp Machine.

3. BOUWSTEEN

BOUWSTEEN (work by Huls) is meant to be the first step in the grammar curriculum. It aims at making students aware of the constituent structure of sentences; that is, of the fact that sentences can be put together by combining fixed word groups in certain left-to-right sequences. Students thus can develop a notion of word order constraints. In BOUWSTEEN this knowledge is taught in a way completely different from the traditional one. In traditional grammar instruction students usually learn to split up given sentences into parts, e.g. by putting bars between constituents. In BOUWSTEEN they have to build sentences out of parts of speech belonging to all the categories. The word groups are stored in a database. A random selection from them is displayed in a menu on the screen. The student may construct any sentence he likes by selecting (pressing cursor keys) appropriate word groups in a correct order. He is allowed to use as many constituents as he likes. The constituents chosen are displayed at the top window of the screen (Figure 2). The sentence is analyzed by the IPG parser (written by Konst, 1986) in just a few seconds. If the

sentence is grammatically correct, a second screen appears. In the menu displayed on this screen, the student only finds the constituents used in the sentence he put together (Figure 3). With these, he has to build as many additional sentences as possible. The IPG parser checks whether (1) all constituents are used in each new sentence, (2) the sentence is grammatical correct, and (3) the sentence is different from the former one(s). Whenever the student wishes to work with a new set of constituents, because he dislikes the set given (on screen 1) or he has made enough variations (on screen 2), he can ask for another menu. Whenever a student makes an ungrammatical sentence on screen 1 or screen 2, this is signaled by the parser. Then, the ungrammaticality is explained to the student at an appropriate level.

BOUWSTEEN also contains a pop-up menu, which gives the student the possibility to ask for examples and explanation. When the student stops a BOUWSTEEN session, he can get a summary of the subject matter taught. For example, it is explained that the word groups are called zinsdelen ('sentence parts'), and he is shown a rudimentary tree structure in which the hierarchical relationship between a zin (sentence) and zinsdelen (constituents) is visualized (Figure 4). Further extensions of this tree structure will be worked out in future grammar instruction programs. The closed subtrees of Figure 4 will be opened, so that their internal structure will be visualized. In this way, the student is gradually familiarized with all grammatical categories and functions in their mutual dependencies. Finally, students will be able to work with complete tree structures of the kind used in the COGO program (paragraph 4).

The described method for grammar instruction is a constructive one: the students build and rebuild their own sentences. This contrasts with the usual analytic method, where students only have to label constituents of given sentences. A user test with twenty six-students aged 12 has shown that half an hour of training with a preliminary BOUWSTEEN prototype already brought about results comparable to a fully developed analytically oriented grammar instruction program, which covered roughly the same subject matter. We hypothesize that optimal results will be obtained in combinations of constructive and analytic grammar teaching strategies. An example of such a program is COGO.

4. COGO

COGO, which has been developed as a stimulating training environment for syntactic structure analysis, can be used for three different didactic purposes. It enables students to:

1. inspect the structure of newly constructed sentences displayed in a graphical tree notation;
2. present themselves with practice trials on labeling the syntactic functions, constituents and word categories in sentences which they construct out of a vocabulary of words displayed on the screen;
3. to explore the morpho-syntactic properties of words and word classes.

A session in COGO runs as follows. At the beginning, the student finds the screen divided into non-overlapping windows (Figure 5). Every window contains a number of words belonging to one word class. Homographs belonging to different word classes may be inserted in different windows. In the wide horizontal rectangle in the middle of the screen, the student can "write" a new sentence. To do this he moves the mouse to the words he needs for the sentence he has in mind. When clicking on the left button of the mouse, the selected word appears in the rectangle to the right of any words selected earlier. If he needs a word that is not displayed in any window, he can type this new word into the empty rectangle at the top the screen. Then the system requests him to define the syntactic properties of the word (which are needed by the parser) via a sequence of menus. After this, the new word appears in the appropriate window, and the student can use it in his sentence. Clicking on the period or the question mark signals the IPG parser (the same one as used in BOUWSTEEN) to start analyzing the sentence. After just a few seconds the syntactic tree - in IPG terms, with explicit alternation of categories and functions - becomes visible on the screen. The student can initiate a grammatical labeling exercise by asking COGO to hide certain node labels, i.e. to replace them by question marks in the displayed tree. The student decides which type(s) of labels he wants to hide, e.g. all labels, just word class labels, or syntactic function labels at the clause or word group levels. In Figure 6 the student has asked COGO to hide syntactic functions and word classes. After

clicking on a question mark, a menu pops up displaying a small list of labels. The labels listed depend on the place of the question mark in the tree. There are four different menus: syntactic functions at the clause level, word groups (phrases), functions at the word group level, and word classes. The student can select a label by moving the mouse. Short definitions of the selected label appear at the bottom of the screen. When the student has indicated his choice by pushing the appropriate mouse button, COGO compares the selected label with the label computed for that node by the parser. In case of a correct choice, the question mark is replaced by the selected label. If not, an error message appears on the screen. In its present form, COGO cannot yet give any further feedback. After the student has thus removed all question marks, or if he likes to stop with a particular sentence, he can click on the word "AF!" (=done!). The tree then disappears and the lexicon windows are displayed again. In case of ambiguity, COGO displays more than one tree, and the student can select the tree that he considers appropriate.

We have used COGO in an informal evaluation study. We observed that students aged 14-15 found the graphical IPG notation very easy to use, and that they were motivated by the opportunity of devising their own sentences.

5. TDTDT

A third domain of application of our intelligent teaching environment is a problematic aspect of Dutch spelling: the conjugation of verbs. We designed a complete representation of the morpho-syntactic rules which make up the domain of knowledge, and developed modules enabling students to practice their domain knowledge, for automatically diagnosing the errors students make, and an interactive user interface.

Dutch spelling is guided by two main principles: the phonological principle (write a word as it is pronounced in standard Dutch) and the morphological principle (spell related forms the same way). Unfortunately for learners of Dutch, these principles are partially conflicting. Especially in the case of some verbal inflectional endings, a lot of confusion arises (Figure 7). Generally, errors involve an improper application of the phonological principle instead of the

morphological principle in the spelling of verbal inflectional endings. E.g. learners are often at a loss whether a conjugational ending sounding like /t/ is written <dt>, <d> or <t>. The correct solution involves the application of a number of syntactic, morphological, phonological and spelling rules.

The domain expert of TDTDT contains the necessary linguistic knowledge, implemented as a system of concepts in the KRS knowledge representation system (Steels, 1986), to conjugate any Dutch verb. To these linguistic concepts, explanatory information was attached, and a rule-history was added to concepts representing computed verb forms. A rule-history lists the different decisions taken and rules applied to produce a particular word form, and is used in diagnosis.

The domain knowledge is presented according to the algorithmic rule method (Assink, 1983), a recent development in spelling teaching. In this method, an easily manipulable algorithm is devised, pictured on a card, which allows quick and easy decisions on the spelling of verb forms. First, the learner is introduced to a number of concepts playing a role in the solution of the problem and featuring on the card (tense, finiteness, person, number etc.). In the following stage, these concepts are related to spelling rules by means of the algorithmic decision scheme on the card. Exercises are made with the help of the card until the algorithm is internalized and the card becomes superfluous.

We have automated the second part of this teaching method (practicing the algorithmic decision scheme). We assume that learners working with our system have received class room teaching about the different linguistic concepts used in the algorithm. For each specific conjugation problem, a decision tree is pictured on the screen. The solution of a specific conjugation problem is represented as a walk through the decision tree (Figure 8). Each node represents a question (e.g. 'Is it an action in the past?') to which the learner answers by clicking on a menu item. If a wrong choice is made this is explained to the learner. Teaching material (exercises) can be generated by both the system and the learner. In the former case, either by using the on-line lexical database to select random verbs or by selecting verbs from a 'difficult cases' list explicitly programmed by the teacher. In the latter case, the learner has a free choice of the verb he wants to exercise. The infinitive (dictionary

citation form) of the verb is entered, and the system computes all its conjugated forms. Next, a heuristic (based on a comparison of the phonological and the spelling representations of the verb form) is used to select those verb forms which are likely to cause mistakes. These forms are 'computed' by the learner with the help of the decision tree as described above. Corrective feedback is provided whenever a wrong decision is taken.

After some time, the learner should be able to solve conjugation problems without the help of the decision tree. At this point, exercises are presented without explicit algorithmic support, and the input by the learner is compared to the forms computed by the system itself. Again, exercises may be provided either by the system or by the learner. Whenever the learner makes a mistake (i.e. the form input by the learner and the form computed by the system do not coincide), several things may happen. As a first possibility, the particular exercise may be repeated, this time with the decision tree. This should remind the learner of the decisions to be taken in deriving a correct answer, and may provide additional explanation if at some node in the decision tree an incorrect course is taken. However, a more challenging option is to activate at this point a diagnosis system which looks for the 'bug' responsible for the mistake in the learner's internalized knowledge. Detailed diagnosis of problems is essential to be able to provide relevant additional explanation and practizing material (Breuker & Cerri, 1982). We have developed an experimental version of such a diagnostic procedure which we will describe shortly.

A correct solution to a spelling problem can be represented as a sequence of correct decisions forming the conditions of a number of morphological and spelling rules. E.g. a correct computation of the second person singular of redden (to save) involves the following decisions and rules: finite=yes, past=no, singular=yes, first-person=no, second-person=yes, inversion=no, stem-computation-- rule (results in the form redd), morphological structure-building rule for second person singular (results in redd+t), consonant-degemination spelling rule (results in red+t), result is redt. This sequence fills the rule-history slot of the redt verb form after computation of this form by the domain expert. Rules can be uniformly represented as consisting of a number of conditions and a

number of actions, performed if the conditions apply in a particular context. The source of mistakes can now exhaustively be characterized as follows:

1. At the macro-level: the omission or insertion of rules or the muddling up of the correct sequence of the rules.
2. At the micro-level: the omission of conditions or actions, the insertion of conditions or actions or the muddling up of the correct application order of conditions or actions. In our case, conditions are either decisions in the decision tree or conditions based on the form of the verb to be conjugated.

It will be clear from this that if we want to check all possible sources of an error, a combinatorial explosion of alternative possibilities results, even for relatively simple forms. Furthermore, the insertion of extraneous rules, conditions or actions is open-ended, and therefore unpredictable. This proves that an approach involving exhaustive diagnosis (like BUGGY for arithmetic; Brown & Burton, 1978) cannot be transported to more complex domains.

A solution to this problem commonly adopted is the compilation of a bug catalogue (e.g. Anderson & Reiser, 1985), based on an empirical study of the errors learners make, and on the relation of the results of this study to the different rules used by the domain expert. By means of the information thus acquired, it is possible to construct heuristic rules which guide the diagnosis process. However, compiling such a list is a time-consuming activity, and we believe that it can be automated to a large extent through progressive refinement of heuristic diagnosis rules acquired through interaction with learners.

Heuristics always treat a number of potentially relevant aspects of the problem (in this case diagnosis) as irrelevant. This can be justified if the number of things that actually go wrong in practice is considerably smaller than the number of things that can go wrong in principle. We believe this to be the case in the diagnosis of mistakes by learners in general and in the conjugation of verbs in particular.

We start from the rule history computed by the system. As a first hypothesis, the system assumes that the mistake is due to a wrong decision somewhere. All decisions in the list are negated in turn, and the effect of this on the computation of

the verb form is considered. Those decisions, which -- when negated -- result in the same mistake as was made by the learner, may possibly have caused the mistake. Note that the computation of the effect of the negation of a decision often involves the postulating of other decisions. E.g. negating past=no generates some new decisions such as weak=yes (or no). The system assumes the correct decisions to be taken. If several possible causes remain, the system tries to choose one by asking additional questions (E.g. 'Is it a weak verb?') or by generating additional exercises. These additional questions and exercises are by no means unnatural to the learner because they take the same form as the 'normal' questions and exercises while interacting with the system.

The number of possibilities can then be constrained by computing the intersection (if the answer was again wrong) or the difference (if the answer was correct) of the relevant lists of decisions. If no hypotheses remain, i.e. all decisions taken by the learner are considered correct, the bug must be due to a mis-application of one of the rules. Roughly the same method can then be used to identify the rule in question. From such a diagnostic interaction, a heuristic rule is derived. I.e. the association of a symptom (the wrong answer) with a diagnosis (a wrong decision or wrongly applied rule). Symptoms should be expressed at a suitable level of abstraction: it would not be very useful to have a specific verb as symptom. Rather the symptom is expressed in terms of the category of the verb (regular, semi-irregular or irregular classes) and the ending of the stem. In additional interactions with the same learner, already existing heuristic rules are gradually refined.

Our approach to rule-refinement is based on the approach to learning in second generation expert systems (Steels, 1985; Steels & Van de Velde, 1985). Second generation expert systems consist of a heuristic rule system (much the same as traditional expert systems) called the surface model, and an additional deep model which provides an understanding of the complete search space over which the heuristics operate. A second generation expert system can fall back on deep reasoning when its heuristic rules fail. The most important advantage of this architecture lies in the fact that it allows automatic learning of new heuristic rules. Rule learning happens by refinement (non-monotonically): a new heuristic

rule is abstracted (in our case from an interaction with the learner), and is integrated into the already existing rule-base. This integration does not make the previous rules superfluous, but may restrict their application. Two options are open in the use of heuristic rules: either a new rule base is constructed for each learner, or the same is used for all learners. In the latter case, the identity of the learner may be added to the list of symptoms.

An important facet in the development of ICAI systems is the user interface. The program should be easy to work with for children, and attractive enough to motivate the learner to work with it. In TDTDT, the decision tree is built up in interaction with the learner. At each node, a question is asked, and the answer (mostly yes or no) is given by clicking with the mouse on a menu of options. If the answer is correct, a branch and new node are drawn, and a new question is asked. If the answer is wrong, the learner is told so, and some explanation is given.

In an exercise (the computation of the correct form of a verb), the infinitive of a verb is presented in a syntactic context. The generating of these contexts is still a problem. Only limited information about the case frame associated with the verb is available from the lexical database, and no semantic information at all. Contexts are therefore very simple and canned phrases designed to be applicable to (almost) all verbs. The problem can be partly overcome by providing a more detailed and natural context and letting the learner choose an applicable verb himself (this limits his free choice, but not to a point where only one or a few possibilities are left open).

An alternative solution would be to explicitly ask for a specific form (e.g. 'What is the inflected past participle of the verb werken (to work)?). In the latter case, a lot of the decisions that must be taken in the real-life computation of verb forms are given in the question, which diminishes the functionality of the tutoring environment considerably.

6. CONCLUSION

Intelligent teaching environments like BOUWSTEEN, COGO and TDTDT may become important steps in the liberation of class

teaching from the less interesting aspects of the subject matter and in the adaptation of the instruction to the needs and pace of individual learners. We have shown that such systems can be developed even for subject matters traditionally considered 'complex'. It should be emphasized that the described instructional tools rely on sophisticated hardware and software resources (with the possible exception of BOUWSTEEN, which runs on an IBM-AT with 3 MByte of internal memory), presently out of reach for most schools. However, we are confident that the evolution of the microcomputer market will make possible the implementation of similar programs on cheaper systems.

REFERENCES

Anderson, J. and Reiser, B. (1985). The LISP TUTOR. Byte, 10-4, 1985.

Assink, E.M.H. (1983). Leerprocessen bij het spellen: aanzet voor de verbetering van de werkwoordsdidaktiek. Dissertation University of Utrecht.

Breuker, J and Cerri, S. (1982). A new generation of teaching machines: intelligent and rather-intelligent computer assisted instruction discussed and exemplified. In E. van Hees and A. Dirkzwager (Eds.), Onderwijs en de nieuwe media. Lisse: Swets and Zeitlinger.

Brown, J.S. and Burton, R.R. (1978). Diagnostic models for procedural bugs in basic mathematical skills. Cognitive Science, 2, 155-192.

Daelemans, W. (1987). Studies in Language Technology. An Object-Oriented Computer Model of Morphophonological Aspects of Dutch. Dissertation University of Leuven.

Desain, P. (1985). TREE DOCTOR, a software package for graphical manipulation and animation of tree structures. In Proceedings of the Symposium on Man-Machine Interaction, Amsterdam, December 1985.

Kempen, G., Schotel, H. and Pijls, F. (1985). Taaltechnologie en Taalonderwijs. In J. Heene and Tj. Plomp (Eds.), Onderwijs en Informatietechnologie. Den Haag: SVO.

Kempen, G., Anbeek, G., Desain, P., Konst, L. and De Smedt, K. (1987). Author Environments: fifth generation text processors. In Esprit '86, Proceedings of the ESPRIT 1986 Conference. Amsterdam: North-Holland.

Kempen, G. and Hoenkamp, E. (1987). An Incremental Procedural Grammar for sentence formulation. Cognitive Science, 12, 201-258.

Konst, L. (1986). A syntactic parser based on filtering. Paper, University of Nijmegen.

Schotel, H. and Pijls, F. (1986). Een prototype van grammatica-onderwijs op een LISP-machine. Informatie, 28-1, 48-50.

Shneiderman, B. (1982). The future of interactive systems and the emergence of direct manipulation. Behavior and Information Technology, 1, 237-256.

Steels, L. (1985). Second generation Expert Systems. In Future Generation Computer Systems. Amsterdam: North-Holland Pub.

Steels, L. (1986). Tutorial on the KRS concept system. Paper, AI-Lab, Free University Brussels.
Steels, L. and Van der Velde, J. (1985). Learning in Second Generation Expert Systems. In J. Kowalik (Ed.), Knowledge Based Problem Solving. New Jersey: Prentice-Hall Inc.

toegepaste

ONDERWIJSKUNDE

Een publicatie van de
Onderzoeksthemagroep Onderwijsleerprocessen
Secretariaat: Dr. J.M. Pieters
Faculteit der Toegepaste Onderwijskunde
Universiteit Twente
Postbus 217, 7500 AE Enschede 82